



# 30 Days Of Metal

**- WARREN MOORE**

***Unofficial Compilation***

30 Days Of Metal - WARREN MOORE Unofficial Compilation	1
Day 1: Devices	8
Welcome to Metal!	8
Devices	9
Getting a Device	9
Day 2: Buffers	11
Data?	11
Creating a Buffer	11
Copying Data into a Buffer	12
Day 3: Commands	15
What are Commands?	15
Queuing Up	16
Creating and Submitting Command Buffers	16
Encoding Commands	20
Day 5: Shaders	38
What is a Shader?	38
The Metal Shading Language	39
Vertex Functions	41
Fragment Functions	43
Libraries	43
Conclusion	46
Day 6: Pipelines	47
Kernel Functions	47
Creating Compute Pipelines	48
Organizing Compute Work	50
Encoding Compute Work	51
Day 7: Drawing in 2D	56
The Graphics Pipeline	56
Coordinate Spaces, Briefly	59
Normalized Device Coordinates	59
A Renderer Class	60
Render Pipeline States	63
Preparing the Vertex Buffer	66
Encoding Draw Calls	66
Day 8: Vertex Attributes	72
Attributes: Beyond Vertex Positions	72
Alignment Considerations	73
Vertex Descriptors	75
Adapting Shaders to Use Vertex Descriptors	78
Day 9: Constants	81

Constant Data	81
Getting Constants into Shaders	81
Triple Buffering	83
Data Synchronization	84
Updating Per-Frame Constants	86
Day 10: 2D Math	89
Points and Vectors	89
Point and Vector Arithmetic	91
Translating Points	93
Scaling Points	94
Rotating Points	97
Composing Transformations	98
Enter the Matrix	99
Matrix Multiplication	99
Expressing Transformations as Matrices	100
Virtualizing the Canvas	103
Projection Transformations	104
Normalized Device Coordinates, in Depth	104
Orthographic Projections	106
Matrix Math in Swift	106
Matrix-Vector Math in Shaders	108
Animating Transformations in Time	108
Day 11: Meshes	115
Primitives, Connectivity, and Meshes	115
A Simple Mesh Class	118
Generating Meshes Programmatically	120
Updating the Renderer to Draw Meshes	124
Indexed Geometry	129
Indexed Geometry in Metal	130
Generating Indexed Geometry	132
Indexed Draw Calls	134
Day 12: MDLMesh and MTKMesh	138
Introducing Model I/O and MDLMesh	138
Model I/O Buffer Allocation	139
MetalKit Buffer Allocation	140
Creating Meshes with Model I/O	141
Bringing Model I/O Data into MetalKit	143
Adapting the Vertex Function	144
Adapting the Fragment Function	146
Adapting the Render Pipeline	146

Drawing an MTKMesh	147
Revising the Projection Matrix	149
Conclusion	149
Addendum	150
Day 13: Depth	151
Thinking Like a Painter	151
Depth Buffering	152
The Depth Buffer in Metal	152
Depth-Stencil States	153
Winding, Facing, and Culling	155
Day 14: Perspective	159
Model Space	159
World Space	160
View Space	161
That W Coordinate	162
Clip Space	162
The Perspective Divide	165
Perspective Projection	165
Day 15: Hierarchy	170
Representing Hierarchy: A Node Class	170
Bundling Constant Data	173
Building a Hierarchy of Nodes	174
Allocating Space for Constants	175
Computing the View Matrix	176
Computing Transformation for Orbital Dynamics	178
Updating Dynamic Constants	180
Updating the Shader Functions	183
Updating the Draw Method	183
Day 16: Textures	186
Textures in Metal	186
Creating a Texture	187
Loading a Texture from an Asset Catalog	190
Texture Space and Texture Coordinates	192
Texture Mapping	192
Generating Texture Coordinates with Model I/O	193
Sampling	194
Address Modes	195
Sampler States	197
Using Samplers and Textures in Shaders	199
Binding Samplers and Textures	201



Day 17: Assets	204
Getting 3D Models	205
Introducing MDLAsset	205
Day 18: Directional Light	214
A Light Class	215
Introducing Multiple Constant Data Types	216
Updating Constant Data	221
Ambient Light	225
Specular and Diffuse Reflectance	225
Theory of Diffuse Reflectance	226
Theory of Specular Reflectance	228
Updating the Vertex Shader	232
Implementing a Lighting Fragment Function	234
Day 19: Directional Shadows	239
Positioning and Orienting Lights	240
Look-At Transforms	241
Shadow Projection Matrices	243
Creating Depth Textures	245
Multipass Rendering	247
Render Pass Descriptors for Shadow Mapping	249
Shadow Mapping Vertex Function and Render Pipeline	249
Using Shadow Maps	251
Shadow Acne	255
Day 20: Multisample Antialiasing	257
Multisample Antialiasing	258
Sample Counts and Positions	259
MSAA with MTKView	259
Creating MSAA Render Targets Manually	261
Creating MSAA Render Pass Descriptors	265
Day 21: Point Lights	271
Attenuation	272
Implementing Attenuation	273
Point Lights in Action	275
Day 22: Instancing	278
The Sample Scene	278
Loading Mesh Assets	279
Creating Node Batches	280
Updating Per-Instance Constants	283
Instancing in the Vertex Function	285
Instanced Draw Calls	287

Day 23: Interaction	292
Camera Controllers	292
Fly Cameras	293
Mouse Input on macOS	299
Keyboard Input on macOS	301
Mapping Keyboard and Mouse Inputs	302
The GameController Framework	303
Virtual Controllers on iOS	307
Day 24: Transparency	310
The Alpha Channel	311
A Theory of Composition	311
Premultiplication	313
Alpha Blending	314
Alpha Blending in Metal	317
Order Dependency in Blending	319
Day 25: Environment Mapping	322
Cube Maps	322
Environment Maps	333
Environment-Mapped Background	335
Reflection	340
Refraction	343
Day 26: Normal Mapping	345
Normal Mapping	345
Tangent Space	346
Normal Map Generation	347
Generating Tangents with Model I/O	348
Tangent Space and Texture Space	350
Implementing Normal Mapping in Metal	350
Day 27: Tessellation	356
Tessellation Factors	358
Authoring Meshes for Tessellation	359
A Patch Mesh Class	359
Tessellation in Metal	360
Tessellation Factors	360
Patch Draw Calls	362
The Post-Tessellation Vertex Function	365
Displacement Mapping	368
Day 28: Skinning	373
Rigging and Skinning	374
Skeletal Animation	375

Skeletons in Model I/O	376
A Skeleton Class	378
Vertex Skinning in Metal	380
Skeletal Animation in Model I/O	384
Animation Playback	389
Finding Nodes	392
Root Motion	393
Day 29: Physically Based Rendering	395
Material Models for Physically Based Rendering	396
Metalness	397
Roughness	397
Base Color	397
What's Missing from Our Material Model	398
Theory of Physically Based Rendering	398
The Microfacet Model	399
Bidirectional Reflectance Distribution Functions	401
The Specular BRDF	402
Remapping Roughness	402
The Normal Distribution Function	403
The Shadowing-Masking Function	405
Fresnel Reflection	408
Assembling the Specular BRDF	409
The Rendering Equation	409
Physically Based Materials in Model I/O	410
Physically Based Shader Preliminaries	413
Implementing the Diffuse BRDF	414
Implementing the Specular BRDF	416
Evaluating the Total BRDF	418
Results	420

# Day 1: Devices



Warren Moore

3 min read · Apr 2, 2022

*This series of posts is my attempt to present the Metal graphics programming framework in small, bite-sized chunks for Swift app developers who haven't done GPU programming before.*

*To run the sample code that accompanies these posts, you will need a fairly recent Mac with Xcode 13 installed. Most of the code can run on an iPhone with relatively little modification, but an iPhone is not required.*

## **Welcome to Metal!**

What exactly is Metal, and why should you care?

Metal is a *framework*—a collection of application programming interfaces (APIs) — that enables you to program the graphics processing unit (GPU) in your phone or computer.

One evocative use of GPU programming is to render 3D graphics for video games, so this series on Metal will cover topics in 3D graphics programming that are essential for real-time interactive apps like games. You should be aware, though, that GPUs are used for other things besides games, like machine learning, data visualization, and physical simulations.

Unlike other graphics frameworks (Core Graphics, SceneKit, SwiftUI Canvas, etc.), Metal requires you to have more low-level knowledge. This entails understanding how to write code that runs on the GPU and how to explicitly describe the work that goes into drawing your pictures. My task with this series is to make that job approachable and at least a little bit fun.

Over the next 30 articles, we will look at many aspects of graphics programming, culminating in sample code that renders an animated, interactive, three-dimensional scene. With the knowledge you gain, you will be able to add 3D graphics to your own apps and games and be well on your way to GPU mastery.

## ***Devices***

To get started, we will learn about devices.

A device is an abstraction of the graphics processing unit, or GPU, inside your iPhone or Mac. The GPU is a separate processor from the CPU and is specialized for different kinds of work. A big part of our job here is learning what GPUs are good at and how to program them.

In code, a device is an object that conforms to the `MTLDevice` protocol. This protocol includes methods for allocating GPU resources, as well as many other kinds of objects. We will see many of these as we progress through our exploration of Metal.

## ***Getting a Device***

I recommend creating a Playground in Xcode and running the code in this article as we go along.

The first thing you should do is import the Metal framework so the compiler can find Metal's types and functions:

```
import Metal
```

The simplest way to get a Metal device is to call the `MTLCreateSystemDefaultDevice()` function. As its name suggests, this function returns the default device. Its return type is `MTLDevice?`, since it can return `nil` on systems that do not support Metal.

```
let device = MTLCreateSystemDefaultDevice()!
```

Once we have a device, we can print out its name:

```
print("Device name: \(device.name)")
```

On iOS devices, there is only one Metal device, while some Macs have several. Here are some possible device names you might see if you run this code on a Mac or iPhone:

```
AMD Radeon Pro 5500M  
Intel(R) UHD Graphics 630  
Apple A10 GPU  
Apple A14 GPU
```

Asking for the name of a device may sometimes be useful, but that's not the most interesting thing we can do by far. In the next article, we will start talking about *resources*, objects that hold the data that the GPU operates on.

# Day 2: Buffers



Warren Moore ·

3 min read · Apr 3, 2022

In the previous article we got acquainted with Metal and learned a little about devices.

In this article, we will start to allocate memory on the GPU in the form of *buffers*. Buffers are essential to graphics programming, because they hold the data that the GPU operates on.

## Data?

That's a bit abstract, so let's talk about *what kinds* of data might be held in a buffer.

In some graphics APIs, if you want to draw a line, there might be a function called `drawLine` to draw a line segment, or perhaps a pair of functions called `moveTo` and `lineTo`, to specify where a line starts and ends.

In Metal, we don't have such convenient APIs available. Instead, to draw a line, we need to store the line's endpoints in a buffer, then issue commands to the GPU that tell it to draw lines based on the data in that buffer.

In the next article, we will start to look at *how* we prepare these commands to be executed by the GPU, but the important point for now is: if you want to draw anything with Metal, the information needed to do so must be stored in memory that the GPU can access. And in Metal, that means that it must be in a buffer.

## Creating a Buffer

We create buffers in Metal by requesting them from our device using the

`makeBuffer(length:options:)` method and its siblings:

```
let buffer = device.makeBuffer(length: 16, options: [])!
```

The `length` parameter is the size of the buffer's memory in bytes (16 bytes, in this case). The `options` parameter allows us to control some aspects of the buffer's creation, but for now we leave it empty.

To verify that the buffer has the expected size, we can print it out:

```
print("Buffer is \(buffer.length) bytes in length")
```

## ***Copying Data into a Buffer***

We won't be able to do much with our buffer unless we copy some data into it. Continuing the example of line drawing from above, let's learn how to put the endpoints of a 2D line into our new buffer.

We first get a *pointer* to the buffer's memory by calling its `contents()` method. This method returns an `UnsafeMutableRawPointer`, which tells us that Swift has no idea what type or how much memory is held by the buffer. That's okay; we know what type of data we want to put into it.

To add type information to the pointer, we can *bind* it to a particular type. The most natural Swift type for storing two-dimensional points is `SIMD2<Float>`, a vector type containing two floating-point coordinates. We bind our buffer pointer to this type, indicating that it has room for two points:

```
let points = buffer.contents().bindMemory(to: SIMD2<Float>.self,  
                                           capacity: 2)
```



Binding the memory makes it possible to treat `points` much like an

ordinary Swift array. Assigning elements of this “array” copies the data directly into the buffer. Let’s write a pair of points into the buffer:

```
points[0] = SIMD2<Float>(10, 10)
points[1] = SIMD2<Float>(100, 100)
```

To verify that we have successfully written into the buffer, we can retrieve the second point and print its value:

```
let p1 = points[1]
print("p1 is \(p1)")
```

On my system, this prints

```
p1 is SIMD2<Float>(100.0, 100.0)
```

indicating success!

At this point, you might be feeling like we’re moving pretty slowly, like we’re never going to get around to actually “doing graphics.” I want to assure you that this is normal. Learning to use Metal takes a lot longer than just picking up a high-level API and running with it. However, knowing Metal allows you to do things you could never do with those APIs. So take heart, because in the next article, we’ll start looking at how to put the GPU to work.

# Day 3: Commands



Warren Moore ·

5 min read · Apr 4, 2022

*This series of posts is my attempt to present the Metal graphics programming framework in small, bite-sized chunks for Swift app developers who haven't done GPU programming before.*

*If you want to work through this series in order, start [here](#).*

So far in this series, we have talked about devices and buffers. Creating a device and asking it to allocate GPU memory are important tasks, but in order to put the GPU to work, we need to learn how to speak its language. We need to learn how to issue commands.

## ***What are Commands?***

As mentioned in the previous article, Metal doesn't have functions that let us simultaneously supply data and draw shapes. Providing data and issuing draw commands are separate operations, and they comprise the most common types of Metal commands.

Let's make this a little less abstract. Say you have a buffer that contains a couple of points, and you want to draw a line segment. First, you would tell Metal *which* buffer holds the line data. Then, you would tell Metal to draw the line. In pseudocode, it might look like this:

```
commandList.setBuffer(pointBuffer)
commandList.drawLines(1)
```

This isn't real Metal code, but it encapsulates an extremely common

pattern. First, supply the data you want the GPU to operate on, then specify the operations you want to carry out.

## Queuing Up

The GPU is a separate processor from the CPU. We want these two processors to be able to run simultaneously, rather than one waiting on the other to be able to do its work.

For this reason, the GPU doesn't immediately execute instructions as we specify them. Instead, commands are collected in *command buffers*, then shipped off to the GPU as a batch.

We deliver commands to the GPU using an object called a command queue. In code, a command queue is an object that conforms to the `MTLCommandQueue` protocol. A command queue's purpose in life is to create command buffers we can populate with commands and deliver them to the GPU when they're ready to execute.

We create a command queue by asking for one from our device:

```
let commandQueue = device.makeCommandQueue()!
```

Like our device, this command queue will stay alive for the duration of our app's execution; we only ever need to create one queue.

## Creating and Submitting Command Buffers

Of course, a queue isn't much use if it's empty forever. So let's give our queue some work to deliver to the GPU. Recall that we put commands into a command buffer that is then delivered to the GPU by the command queue.

We create a command buffer by asking for one from the queue:

```
let commandBuffer = commandQueue.makeCommandBuffer()!
```



At this point, we have a command buffer that we can write commands into, commands like “use this buffer” and “draw a line” and “copy some data from here to there.” However, the command buffer itself doesn’t have methods that let us specify these commands; that’s done by a separate kind of object we’ll look at next.

Instead, for the moment, we’ll just tell the GPU to run our (empty) command buffer and let us know when it’s done.

We can know when a command buffer is done by adding a completed handler to it:

```
commandBuffer.addCompletedHandler { completedCommandBuffer in
    print("Command buffer completed")
}
```

Getting notified of command buffer completion asynchronously allows the CPU and GPU to continue running independently, which we always want, for maximum efficiency.

To actually execute the command buffer, we commit it, which lets its associated command queue know it’s ready for execution:

```
commandBuffer.commit()
```

If you run the code above in a Swift Playground, you should see the following output:

```
Command buffer completed
```

This indicates that we've successfully created a command queue, used it to enqueue a command buffer, and been notified of its completion. That may not feel like much progress, but it actually is: every app that uses Metal uses

this command submission model.

Now we just need to get some commands into the buffer.

## ***Encoding Commands***

We call the process of writing commands into a command buffer *encoding*. We'll start drawing things soon enough, but the easiest way to introduce command encoding is with the blit encoder.

The main purpose of the blit encoder is to efficiently copy regions of memory between resources (i.e., buffers and textures).

Suppose we want to copy data from our buffer of line segment points to another buffer. We might start by creating our “source” and “destination” buffers, much as we did in the previous entry.

```
let sourceBuffer = device.makeBuffer(length: 16, options: [])!  
let destBuffer = device.makeBuffer(length: 16, options: [])!
```

Also as before, we can write point data into the source buffer by forming a mutable pointer to a `SIMD2<Float>` array and setting its elements:

```
let points = sourceBuffer.contents().bindMemory(to:  
SIMD2<Float>.self,  
                                                capacity: 2)  
points[0] = SIMD2<Float>(10, 10)  
points[1] = SIMD2<Float>(100, 100)
```

To copy data between buffers, we will use a blit command encoder. Assuming we already have a device, command queue, and command buffer ready to go, we can ask our command buffer for a blit encoder:

```
let blitCommandEncoder = commandBuffer.makeBlitCommandEncoder()!
```





Each command encoder type has methods that encode its respective kind of operations: compute command encoders encode commands related to general-purpose computation on the GPU, render command encoders encode rendering (drawing) commands, and blit command encoders encode copy commands.

We will use the `copy(from:sourceOffset:to:destinationOffset:size:)` method to copy from one buffer to another. Since we want to duplicate the data completely between our source and destination buffers, both offset parameters will be 0, and the size to copy will be twice the stride of one of our points, or 16 bytes in total.

```
blitCommandEncoder.copy(from: sourceBuffer,
                        sourceOffset: 0,
                        to: destBuffer,
                        destinationOffset: 0,
                        size: MemoryLayout<SIMD2<Float>>.stride * 2)
```

This method writes the copy command into the encoder's associated command buffer. It does not cause the copy to happen immediately; it happens after we commit the command buffer.

When we're done writing commands with a command encoder, we call `endEncoding()` on it.

```
blitCommandEncoder.endEncoding()
```

As before, we can then add a completed handler before committing the command buffer, so we know when the copy command is done executing.

To verify that our copy command actually worked, we can use the completed handler to retrieve the point data from our destination buffer and print it out.

```
commandBuffer.addCompletedHandler { completedCommandBuffer in
```

```
    let outPoints = destBuffer.contents().bindMemory(to:
SIMD2<Float>.self,                                     capacity: 2)

    let p1 = outPoints[1]
    print("p1 in destination buffer is \(p1)")
}
```

Running this code prints the following:

```
p1 in destination buffer is SIMD2<Float>(100.0, 100.0)
```

Obviously this example is pretty trivial. It seems we've done a lot of work just to copy a handful of bytes from one area in memory to another, but consider this example in a broader context. You now know how to encode and enqueue commands for the GPU, and that's an important step toward using Metal to render 3D graphics.

The next step on our Metal journey is the view class that enables us to display rendered content on the screen. In the next article, we'll finally *see* some results of our labor.

# Thirty Days of Metal — Day 4: MTKView



Warren Moore ·

6 min read · Apr 5, 2022

*This series of posts is my attempt to present the Metal graphics programming framework in small, bite-sized chunks for Swift app developers who haven't done GPU programming before.*

*If you want to work through this series in order, start [here](#).*

In the previous article, we got acquainted with Metal's command submission model, including command queues, command buffers, and command encoders. In this article, we will meet the `MTKView` class, which allows us to display the pictures we draw with Metal on the screen.

## Introducing MTKView

The `MTKView` class is provided by the MetalKit framework. MetalKit is a small framework that provides higher-level utility types that make working with Metal easier.

`MTKView` is a subclass of the basic view class in AppKit in UIKit. On macOS, it inherits from `NSView`, while on iOS it inherits from `UIView`. At this time, Apple hasn't provided a SwiftUI-native Metal view, so if you want to put one of these views in a SwiftUI view body, you'll need to wrap it with

`NSViewRepresentable` / `UIViewRepresentable`.

You use `MTKView` just like you would any other view type. You can add it to a storyboard, instantiate it manually and add it to a view hierarchy, etc. The difference is that the contents of an `MTKView` are updated by using Metal to draw things with the GPU.

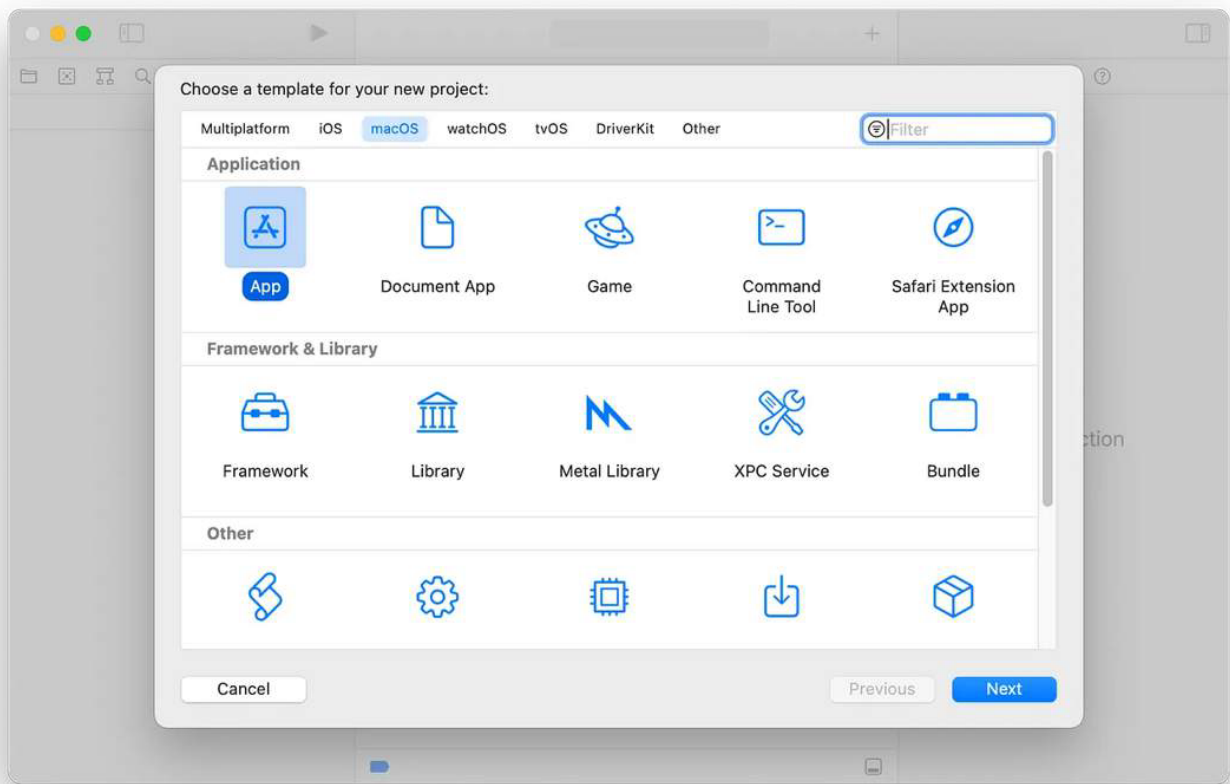
Under the covers, an `MTKView` is backed by a special `CALayer` subtype called `CAMetalLayer`. This Metal layer, in turn, provides *textures*, which are

resources that hold image data.

We'll talk a lot more about textures in subsequent articles. The important part is that whenever you draw things with Metal, you're drawing them *into* a texture, and `MTKView` is the conduit through which those textures are displayed on the screen.

## Creating an MTKView with Storyboards

Let's write some code. This time, instead of working in a Playground, we start by creating a new *project* in Xcode. Use the macOS App template, choose a name, and make sure that you select Swift as the language and Storyboard as the interface paradigm.



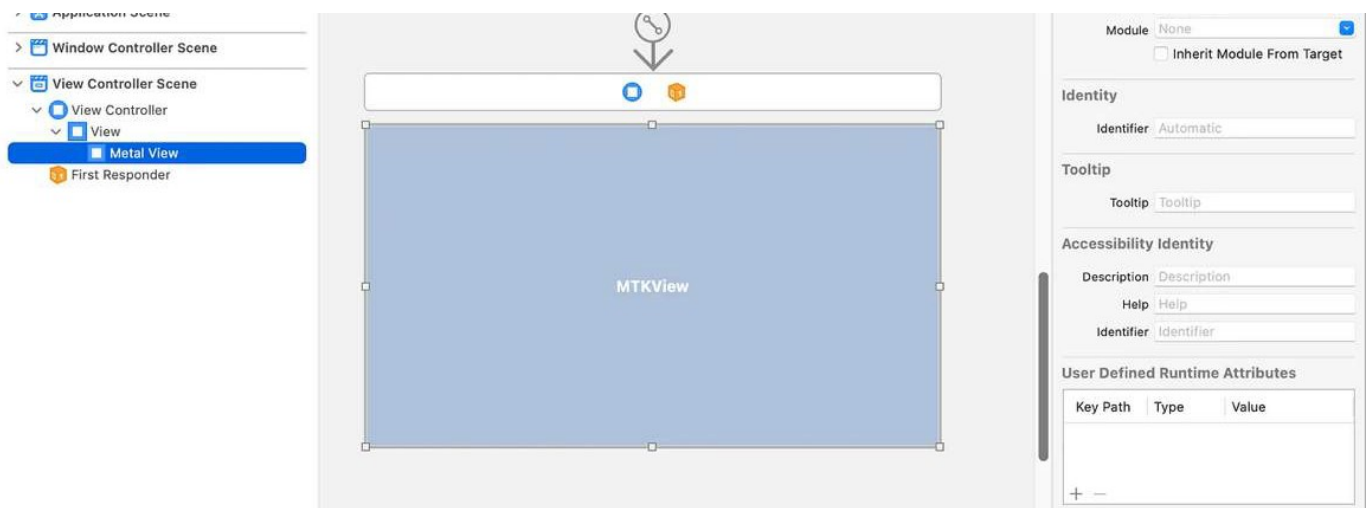
Once the project is created and saved somewhere, open Main.storyboard.

Interface Builder does not include `MTKView` as one of the view types in the object palette. Instead, we drop a Custom View onto the root view of the view controller. We can then configure its class in the inspector. Set it to

`MTKView`.







## Associating a Device with a View

An `MTKView` doesn't draw anything on its own; instead, we have to issue commands to the GPU to tell it to draw for us.

The first step to ensuring we have somewhere to draw is to set the `MTKView`'s device. Associating a device to the view allows the view to create textures on our behalf. These textures then become the "canvas" to which we draw.

To set the view's device, we need a reference to the view, so wire up an outlet in the `ViewController` class that refers to the view. I named mine `metalView`.

We instantiate our device as before: by calling `MTLCreateSystemDefaultDevice()`. We then set the `device` property on our view to this device.

The complete listing for `ViewController.swift` so far is shown below. Make sure the outlet is actually connected to the Storyboard by verifying that the bubble next to its declaration is filled in.

```
import Cocoa
import Metal
import MetalKit

class ViewController: NSViewController {

    @IBOutlet weak var metalView: MTKView!
    var device: MTLDevice!

    override func viewDidLoad() {
        super.viewDidLoad()

        device = MTLCreateSystemDefaultDevice()
        metalView.device = device
    }
}
```





```
}
```

Note that we have also added a property to the view controller that stores the device; this is so we can create other Metal objects as necessary down the line.

Building and running the app at this point would be underwhelming. We'd probably just see a gray window, since we haven't given the view a way to tell us when to draw, nor have we issued any GPU commands to draw into it.

## ***The MTKViewDelegate Protocol***

The most convenient way to get notified when an `MTKView` wants to draw is by setting its `delegate` property. By default, the view runs an internal timer that fires regularly, allowing us to render interactive content.

The first step of setting up a delegate is informing Swift that we want our `ViewController` class to conform to the `MTKViewDelegate` protocol:

```
class ViewController: NSViewController, MTKViewDelegate
```

We can then assign the view controller as the view's delegate in

```
viewDidLoad() :
```

```
metalView.delegate = self
```

If we stop here, though, our program won't compile, because we haven't implemented the required methods of the protocol. There are two of them, and minimal implementations of both are shown here:

```
func mtkView(_ view: MTKView, drawableSizeWillChange size: CGSize) {  
}  
  
func draw(in view: MTKView) {  
}
```



The `mtkView(_:drawableSizeWillChange:)` method is called when the view's size changes, so we can reshape the contents to fit the view if needed.

The `draw(in:)` method is called periodically so we have the chance to redraw the contents of the view with Metal. This is where we will do most of our work.

## ***Clearing the Canvas***

We won't quite get around to drawing shapes in this article; there are still a few more concepts we need to introduce first.

Instead, we will show how to clear the view to a solid color using some of the objects we introduced last time, as well as a new encoder type.

The new encoder type is `MTLRenderCommandEncoder`. As its name suggests, it allows us to encode rendering (drawing) commands into our command buffers. The `MTLRenderCommandEncoder` protocol includes many methods for doing all sorts of rendering work.

We create a render command encoder by asking for one from the command buffer. However, unlike blit command encoder creation, creating a render command encoder requires some extra information: a render pass descriptor.

A render pass descriptor is an object that tells the encoder which texture(s) it is drawing into. In this case, because we're drawing into an `MTKView`, these textures are provided by the view itself. We ask the view for its

`currentRenderPassDescriptor` to get the pass descriptor for the current frame:

```
let renderPassDescriptor = view.currentRenderPassDescriptor
```

This render pass descriptor comes preconfigured with the view's current texture, along with some other state that determines how the existing contents of the texture should be treated. By default, the texture is cleared to the *clear color* of the view, which we can control by setting its `clearColor`

property:

```
// Set the view's clear color to a pleasant shade of blue
metalView.clearColor = MTLClearColor(red: 0.0, green: 0.5, blue:
1.0, alpha: 1.0)
```

As always, we need a command buffer before we can create a command encoder, so the first half of the `draw(in:)` method looks like this:

```
let commandBuffer = commandQueue.makeCommandBuffer()!

guard let renderPassDescriptor = view.currentRenderPassDescriptor
else {
    print("Didn't get a render pass descriptor from MTKView;
dropping frame...")
    return
}

let renderPassEncoder =
commandBuffer.makeRenderCommandEncoder(descriptor:
renderPassDescriptor)!
```

First, we make a command buffer, then we ask for the current render pass descriptor, then we use it to make a render command encoder. If we fail to get a render pass descriptor, we skip the current frame.

Now that we have a render command encoder, we could use it to draw some things. But we don't know how to do that just yet. In the meantime, we'll just end encoding, indicating to the command buffer that the current *render pass* is complete.

```
renderPassEncoder.endEncoding()
```

Even without any drawing commands in the command buffer, the texture will still be cleared. Now we just need to get it onto the screen.

When we draw into an `MTKView`, its texture is wrapped in an object called a `drawable`, which allows us to display it on the screen. We call this process *presentation*. To present the current drawable/texture on the screen, we call the `present(_:)` method on the command buffer:



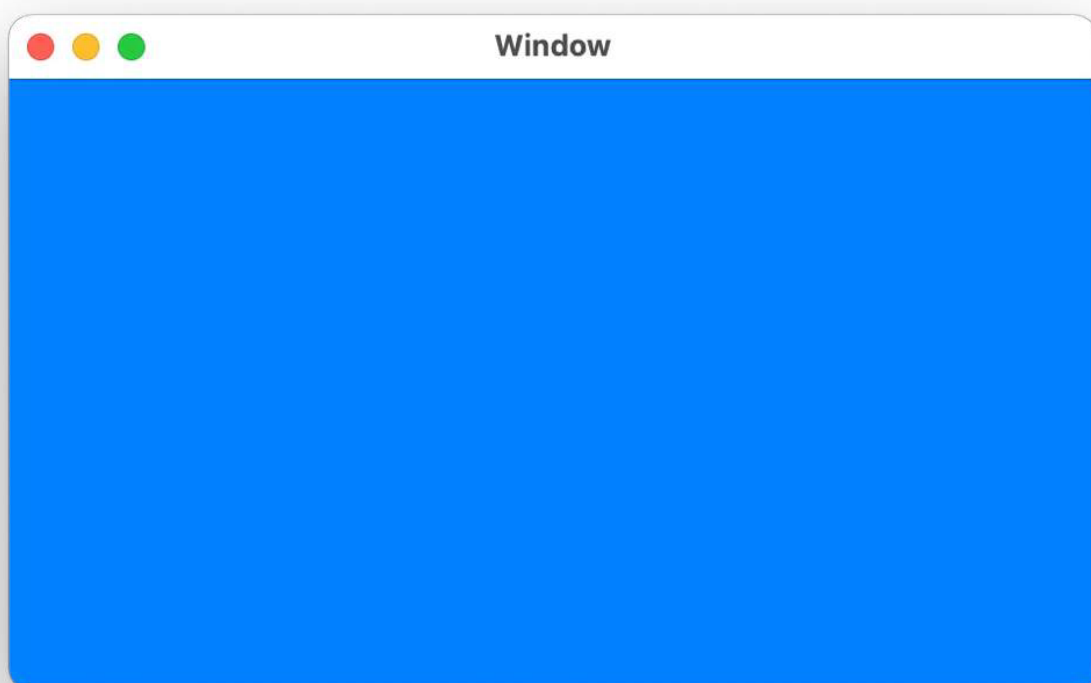
```
commandBuffer.present(view.currentDrawable!)
```

This call tells the system to replace the existing contents of the view with the newly-drawn contents when the GPU is done executing the previous commands (in this case, that just means clearing the texture).

As usual, we can now commit the command buffer and see the results of our work.

```
commandBuffer.commit()
```

If everything went according to plan, you should see the window cleared to a pleasing shade of blue:







All right, now we're getting somewhere! We're finally using the GPU to put colors on the screen. Well, one color. But there will be a lot more colors soon. In the next article, we'll introduce yet another essential topic, *shaders*, and start to write code that runs on the GPU!

# Day 5: Shaders



Warren Moore ·

6 min read · Apr 6, 2022

*This series of posts is my attempt to present the Metal graphics programming framework in small, bite-sized chunks for Swift app developers who haven't done GPU programming before.*

*If you want to work through this series in order, start [here](#).*

In the previous article, we looked at how to start and end render passes by creating a render command encoder from a view's render pass descriptor. We also saw how the act of executing a render pass can clear the contents of a texture to a solid color. Finally, we discussed how to present the cleared texture in the view.

We are now ready to start writing code that runs on the GPU.

## ***What is a Shader?***

We call small programs that run on the GPU “shaders.” When shaders were introduced, the name was apt: the sole purpose of a RenderMan shader was to determine the color of a pixel, to *shade* it.

Nowadays, shaders do more than shading, and this can be a bit confusing. Just keep in mind that when we say shader, we mean “a relatively small program that performs a unit of work on the GPU.”

There are several types of shaders in Metal. The two that we will learn about in this article are *vertex shaders* and *fragment shaders*. The purpose of a vertex shader is to determine where a given vertex (point) should be positioned in space. The purpose of a fragment shader (called a pixel

shader in some other APIs) is to produce the color of a single pixel. We will look at these types of shaders in greater detail as we go along.

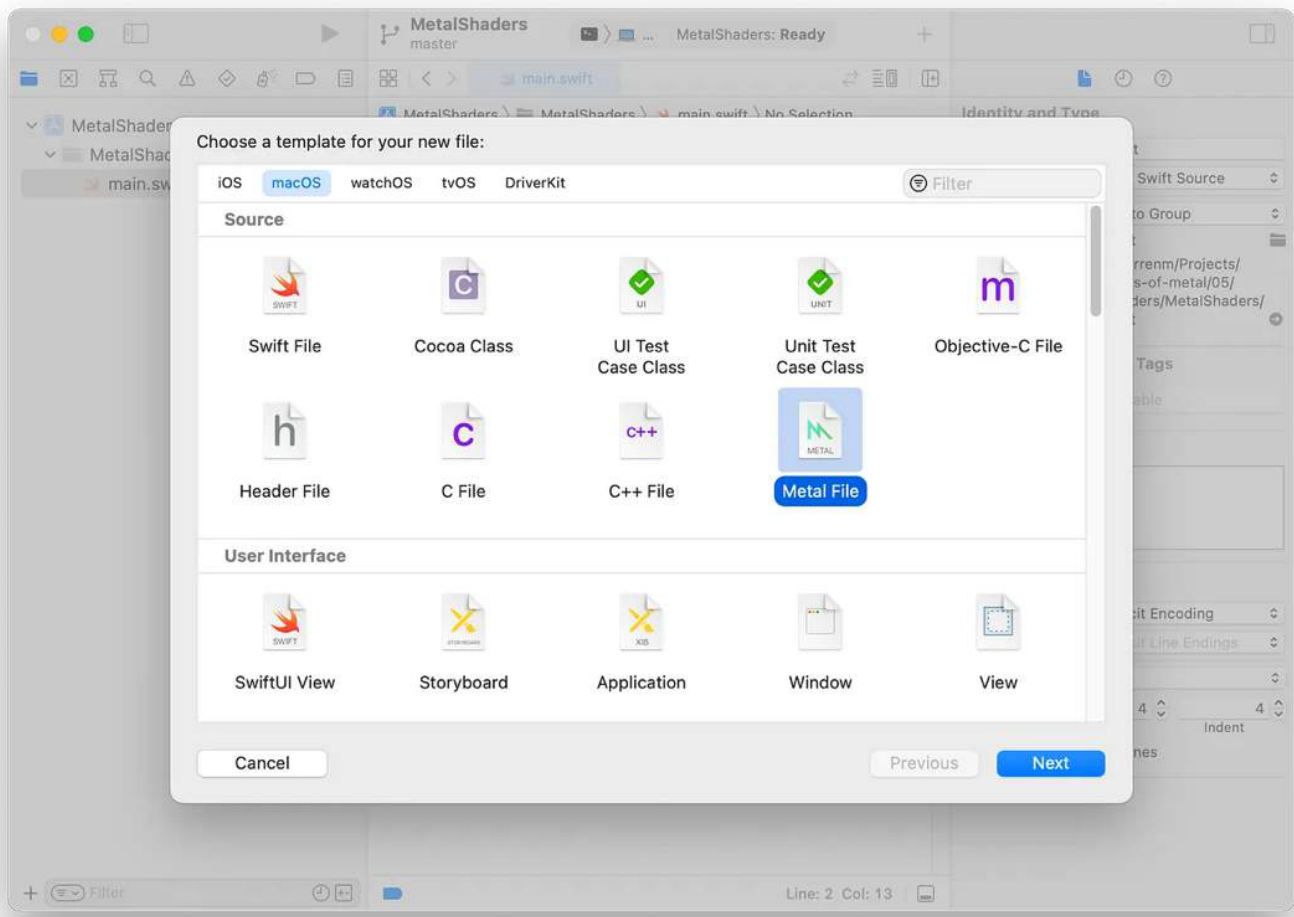
The most general kind of shader is the *compute shader*. These perform arbitrary functions on arbitrary data. Compute shaders can be used alongside render shaders (i.e., vertex and fragment shaders) to implement post-processing effects like bloom, depth of field, and motion blur. They can also be used much more broadly to implement simulations, machine learning, and other types of computation at large scale.

## ***The Metal Shading Language***

You might reasonably assume we would write shaders in the same language as our app: Swift. Unfortunately, this is not the case. Shaders are written in an entirely different language called Metal Shading Language, or MSL.

MSL is a derivative of C++, and its syntax is rather different from Swift's. We will look at a few short examples below, but don't worry if it doesn't stick immediately; learning a new language takes time.

To demonstrate how to write shaders, we can first create a macOS command-line app, which reduces the boilerplate code to a bare minimum. Then, we can create a Metal shader source file using Xcode's New File menu and selecting the "Metal File" template:



This file will have the following lines at the top, which are common to most Metal shader files, and simply import the Metal standard library, making it available throughout the file.

```
#include <metal_stdlib>
using namespace metal;
```

When we start drawing, we will learn about the graphics pipeline, the conceptual flow of data from 3D models to pixels on the screen. One of the first steps, or stages, in that pipeline is vertex processing, which happens partially inside your vertex shader.

When we talk specifically of the code that will be compiled into a shader program, we will use the more precise term *function* or *shader function*. In referring to the function that operates in a particular stage, we will use the even more precise terms *vertex function* and *fragment function*.

Let's take a look at each of those in turn.

## Vertex Functions

The purpose of the vertex function is to fetch data from buffers (called vertex buffers) that contain the geometric data of a single vertex and process it into the final vertex position.

With no further ado, here's a simple vertex function:

```
vertex float4 vertex_main(  
    device float2 const* positions [[buffer(0)]],  
    uint vertexID [[vertex_id]])  
{  
    float2 position = positions[vertexID];  
    return float4(position, 0.0, 1.0);  
}
```

There's a lot to unpack here. How much of this weirdness is due to C++ and how much to Metal? What does the `device` keyword signify? What is that asterisk (`*`) doing? What are those double square brackets?

Don't worry about any of that for now. It'll come in time. Try to see through the syntax to the flow of data: we take a list of positions as a parameter, along with something called a "vertex ID," and we produce a 4-element vector from the input position.

Believe it or not, this little function actually does some useful work. Specifically, it looks up which position corresponds to the current vertex, then transforms it into a format that's useful to the rest of the pipeline.

After a vertex is processed by the vertex function, it goes on to be assembled into a "primitive," or basic geometric shape. A primitive might be a point, a line segment, or a triangle.

A primitive is a kind of conceptual entity. You can't look at a primitive; it only exists as data, a set of points. In order to turn primitives into pixels, we need a special-purpose part of the GPU called the *rasterizer* to chop it up.

The purpose of the rasterizer is to take the primitives we've asked the GPU



to draw and determine which pixels might be a part of them. Then, for each pixel, the fragment function is called to determine its color.

## ***Fragment Functions***

The art of determining pixel colors — *pixel shading* — is an elaborate art. We will get much more familiar with it in later entries, but we'll really only scratch the surface of what's possible.

In the meantime, let's take a look at the simplest possible fragment function: the constant-color fragment shader.

```
fragment float4 fragment_main(float4 position [[stage_in]])
{ return float4(1.0, 0.0, 0.0, 1.0);
}
```

Again, don't worry too much about the syntax. The important thing to note is that this function also returns a 4-element vector. But this vector isn't a position; it's a color. It's the color we want to assign to the current pixel. The elements (*components*) are listed in red, green, blue, alpha (RGBA) order, where alpha represents the opacity of the color.

This fragment function returns a solid red for every fragment. Not very exciting, but we're still just getting started.

I've been somewhat loose in distinguishing between pixels and fragments so far. The difference is subtle but important, but it's easiest to think of fragments as *partial* or *potential* contributors to a pixel's color. One way in which a fragment might not entirely determine a pixel's color is if it's transparent: in that case, the fragment's color is *blended* (mixed) with any existing pixel color to produce its combined color.

So now we've written some Metal shader code. How do we use it?

## ***Libraries***

Like many other kinds of code, shaders must be compiled before they are



run. In Metal, this process happens in two stages. The first stage occurs when your app is compiled, and it produces a file called a *library*. A library file has the extension `.metallib`.

The runtime counterpart of a library file is a library object, and object that implements the `MTLLibrary` protocol. We get a library object by asking our device for it (noticing a pattern yet?)

I prefer to use the `guard let` pattern when creating a library, since it provides the opportunity to report an error. As long as your Metal source is being compiled into your app bundle, creating a library shouldn't fail, but it's good to check.

```
guard let library = device.makeDefaultLibrary() else
    { fatalError("Unable to create default shader library")
}
```

Any Metal shader files in your app target are automatically compiled into a library file called `default.metallib` and copied into your app bundle at compile time. Convenient.

To get references to your shader functions, you request them by name from the library. If you want to inspect a library's contents at runtime, you can use its `functionNames` property to get a list of functions.

In the code below, we iterate the list of functions in our default library, which contains the vertex and fragment function we wrote above.

```
for name in library.functionNames {
    let function = library.makeFunction(name: name)! print("\
(function) ")
}
```

The output will be something like

```
<_MTLFunctionInternal: 0x109706cd0>
  name = fragment_main
  device = <GFX10_MtlDevice: 0x130008000>
  functionType = MTLFunctionTypeFragment

<_MTLFunctionInternal: 0x109707d40>
  name = vertex_main
  device = <GFX10_MtlDevice: 0x130008000>
  functionType = MTLFunctionTypeVertex
```

Each function object (which conforms to the `MTLFunction` protocol but may have a private concrete type) knows its name and its type ( `.fragment` or `.vertex` in this case).

## ***Conclusion***

We've come a long way, but we still have a long way to go. Fortunately, we're very close to being able to actually draw shapes on the screen. Next up, we'll take a deeper look at the graphics pipeline and how we incorporate shaders into larger programs that turn geometry into pixels.

# Day 6: Pipelines



Warren Moore ·

6 min read · Apr 7, 2022

*This series of posts is my attempt to present the Metal graphics programming framework in small, bite-sized chunks for Swift app developers who haven't done GPU programming before.*

*If you want to work through this series in order, start [here](#).*

Let's recap what we've learned in the first five installments of this series.

First, we learned about how to get a device object, which lets us allocate resources and create various other objects related to command submission. Then, we talked about creating buffers, a type of resource that holds the data to be used by the GPU in fulfilling our commands. Then, we talked about how to encode commands and submit them to the GPU for execution. Last time, we started to get acquainted with shaders, the programs we write that run on the GPU itself.

Looking back, we've covered a lot of ground! Our next step is to understand how to build pipelines from our shader functions.

We will begin with the simplest kind of pipeline: *compute pipelines*. In contrast to render pipelines — which have many stages like vertex processing, rasterization, fragment processing, and so on — compute pipelines really only have one stage: “do the work.”

## **Kernel Functions**

The work done by a compute pipeline occurs in a particular kind of shader function called a *kernel function*, also called a “compute kernel” or “compute

function.” Each invocation of a kernel function does a small unit of work on various data to produce an output.

For example, a kernel function might retrieve two numbers from two different input buffers, add them together, then write the result to a third buffer. Here’s what that looks like in code:

```
kernel void add_two_values(constant float *inputsA [[buffer(0)]],
                           constant float *inputsB [[buffer(1)]],
                           device float *outputs [[buffer(2)]],
                           uint index [[thread_position_in_grid]])
{
    outputs[index] = inputsA[index] + inputsB[index];
}
```

Once again, ignore the syntax and instead focus on the operation being performed. We take pointers to two input buffers (`inputsA` and `inputsB`), a pointer to the output buffer (`outputs`), and an index that tells us which element to operate on. In the body of the function, we do the work of getting the two input values, add them, and write the result to the outputs.

But how do we know which index we’re operating on? We’ll get to that, but first we need to know how to construct a compute pipeline that will enable us to use this kernel function.

## ***Creating Compute Pipelines***

Creating a compute pipeline is a two-stage process. Assume we already have a device and a library object (we discussed libraries in the previous entry).

First, we create an `MTLFunction` object referring to the kernel function.

As we saw last time, we can create a function object by asking for it by name from the library:

```
let kernelFunction = library.makeFunction(name: "add_two_values")!
```

We then create the compute pipeline by passing the function to the

`makeComputePipelineState(function:)` method on our device:

```
let computePipeline = try device.makeComputePipelineState(function:
kernelFunction)
```

The `computePipeline` variable now holds a reference to a *compute pipeline state* object, which conforms to the `MTLComputePipelineState` protocol. How do we use such an object?

## Organizing Compute Work

GPUs are built to operate in parallel on many pieces of data at the same time. When we write vertex and fragment functions, we are writing code that might be executed concurrently on dozens of vertices or fragments. This massive concurrency is part of what makes GPUs so efficient.

When we write kernel functions, we aren't necessarily processing vertices or pixels, so we need some other way of organizing work. This is called a *grid*.

A grid is nothing more than a block of work. Grids can be one-, two-, or three-dimensional. You can think of the dimensions of a grid as corresponding to nested `for` loops. A one-dimensional grid is a single `for` loop; a two-dimensional-grid is a nested `for` loop; and a three-dimensional grid is a doubly-nested `for` loop.

In the same way that we might use indices such as `i`, `j`, and `k` to index the iterations of a loop, we uniquely identify the index of a piece of work through its position in the grid.

So how do we define grids in code and use them to get work done?

First, we need to understand that each invocation of our kernel function corresponds to a unique *thread* of execution. We subdivide our grid into groups of threads — called *threadgroups* — that are intended to execute concurrently. Like grids themselves, threadgroups can be one-, two-, or

three-dimensional. The dimensions of our grid are then the size of the threadgroup multiplied by the number of threadgroups we want to execute.

Generally, the number of threads in a threadgroup should be a multiple of 32 or 64 (assuming there are at least that many invocations needed to get the job done). More specifically, it should be a multiple of the compute pipeline state's `threadExecutionWidth` property. Selecting the best threadgroup size is sometimes a matter of experimentation and won't matter for the small examples I'm showing here.

Suppose we have two arrays containing 256 floats each and we want to sum them up. Since arrays are one-dimensional data structures, we might define our threadgroup size like this

```
let threadsPerThreadgroup = MTLSize(width: 32, height: 1, depth: 1)
```

Each threadgroup will have 32 threads, allowing the kernel function to run up to 32 times concurrently.

We now need to select a threadgroup count that, when multiplied by the threadgroup size, equals the total number of array elements.  $256/32=8$ , so we need 8 threadgroups:

```
let threadgroupCount = MTLSize(width: 8, height: 1, depth: 1)
```

Thus our total grid size is  $256 \times 1 \times 1$ .

## ***Encoding Compute Work***

Now that we know how to organize work into grids, let's talk about using our compute pipeline to do the work.

First, let's allocate a few buffers to hold the input and output values:

```

let elementCount = 256
let inputBufferA = device.makeBuffer(length:
MemoryLayout<Float>.stride * elementCount,
                                     options: .storageModeShared) !
let inputBufferB = device.makeBuffer(length:
MemoryLayout<Float>.stride * elementCount,
                                     options: .storageModeShared) !
let outputBuffer = device.makeBuffer(length:
MemoryLayout<Float>.stride * elementCount,
                                     options: .storageModeShared) !

```

Then, let's populate the buffers with some values so we can validate that the operation completed successfully.

```

let inputsA = inputBufferA.contents().assumingMemoryBound(to:
Float.self)
let inputsB = inputBufferB.contents().assumingMemoryBound(to:
Float.self)
for i in 0..

```

Input buffer A contains the numbers 0 to 255 in sequence, while input buffer B contains the numbers 256 to 1 in reverse order. If we sum them up pairwise, each value in the output will therefore be 256.

As we saw previously, we send commands to the GPU in parcels called command buffers. Assuming we have a command queue already, we can ask it to create a command buffer:

```

let commandBuffer = commandQueue.makeCommandBuffer() !

```

We now create a new kind of encoder object: a *compute command encoder*. As the name suggests, we use a compute command encoder to encode compute work for the GPU.

```

let commandEncoder = commandBuffer.makeComputeCommandEncoder() !

```





Unlike the blit command encoder we saw previously, compute and render command encoders require a pipeline state object. This is because render commands and compute commands both execute shader functions, and a pipeline state object contains one or more compiled shader functions.

We set the previously created compute pipeline state on the command encoder before asking it to do any work:

```
commandEncoder.setComputePipelineState(computePipeline)
```

We then set the inputs of our kernel function. Each buffer parameter is “bound” by calling the `setBuffer(:offset:index:)` method. Note that the index parameter of each buffer argument matches the index in the `buffer` attribute in the shader code.

```
commandEncoder.setBuffer(inputBufferA, offset: 0, index: 0)
commandEncoder.setBuffer(inputBufferB, offset: 0, index: 1)
commandEncoder.setBuffer(outputBuffer, offset: 0, index: 2)
```

The command to execute a grid of compute work is called a *dispatch*, so we use the `dispatchThreadgroups(_: threadsPerThreadgroup:)` method to tell the encoder to encode a command to dispatch our grid:

```
commandEncoder.dispatchThreadgroups(threadgroupCount,
                                     threadsPerThreadgroup:
                                     threadsPerThreadgroup)
```

Since that’s all the work we want to encode, we then call `endEncoding` on the encoder:

```
commandEncoder.endEncoding()
```

As before, if we want to see the results of our work, we can print out the elements of the output buffer once we're informed the work is done.

```
commandBuffer.addCompletedHandler { _ in
    let outputs = outputBuffer.contents().assumingMemoryBound(to:
Float.self)
    for i in 0..
```

On my machine, this prints:

```
Output element 0 is 256.0
Output element 1 is 256.0
Output element 2 is 256.0
:
Output element 253 is 256.0
Output element 254 is 256.0
Output element 255 is 256.0
```

Success!

In this article, we learned how to write kernel functions and ask the device to compile them into compute pipeline states. We then learned about how to encode compute work that operates on multiple buffers.

At long last, we're ready to start drawing shapes! Next time, we'll introduce render pipeline states and draw calls, opening a whole new dimension of possibilities.

# Day 7: Drawing in 2D



Warren Moore ·

9 min read · Apr 8, 2022

*This series of posts is my attempt to present the Metal graphics programming framework in small, bite-sized chunks for Swift app developers who haven't done GPU programming before.*

*If you want to work through this series in order, start [here](#).*

Last time, we looked at how to create compute pipeline states from kernel functions so we can perform arbitrary computation on the GPU. This time, we'll focus on a different kind of pipeline state: *render pipeline states*.

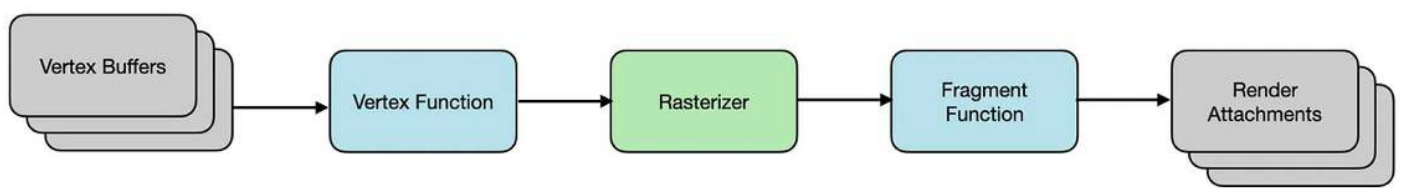
When you have to manage a lot of the moving parts yourself, rendering is a pretty complex task. By now, you probably appreciate just how much more work it is to do anything with Metal than other graphics APIs. Maybe you've also gotten a glimpse of how much more powerful and flexible Metal is as a GPU-oriented API. Or maybe not. But pretty soon, you'll have a much better feel for your latent Metal superpowers.

We will be building on the `MTKView` sample app from Day 4, so consider refreshing your memory on that before continuing.

## ***The Graphics Pipeline***

We often use the phrase “graphics pipeline” to describe the series of stages data flows through to produce digital pictures. At the start of the pipeline, we have a set of geometric data (vertices) that represent the object(s) we want to draw. At the end of the pipeline, we have pixels in a texture.

Here is a greatly simplified diagram of the process:



The light blue boxes in this figure represent “programmable” stages, portions of the pipeline where we are responsible for writing shader code. We have already seen examples of vertex and fragment functions, but have not really begun to use them in earnest.

In brief, the vertex shader reads vertex data and outputs the position of the vertex, along with any other per-vertex data needed by the rest of the pipeline. Vertices are then gathered together into primitives (points, lines, and triangles). Then, the rasterizer determines which pixels belong to each primitive and *interpolates* the values between the vertices. These interpolated vertex properties are fed into the fragment shader, which calculates the color of the fragment. That color is then combined with the existing color in the output color texture. Once this process has run for each vertex, primitive, and fragment, the picture is complete.

The rest of this article will cover what the jobs of the vertex and fragment shader are in more detail, as well as how we build render pipeline state objects from them and use those to encode draw commands.

We will reuse the vertex and fragment shader functions we wrote on Day 5. I’ve included them here for your convenience:

```
#include <metal_stdlib>
using namespace metal;

vertex float4 vertex_main(
    device float2 const* positions [[buffer(0)]],
    uint vertexID [[vertex_id]])
{
    float2 position = positions[vertexID];
    return float4(position, 0.0, 1.0);
}

fragment float4 fragment_main(float4 position [[stage_in]])
{ return float4(1.0, 0.0, 0.0, 1.0);
}
```



## ***Coordinate Spaces, Briefly***

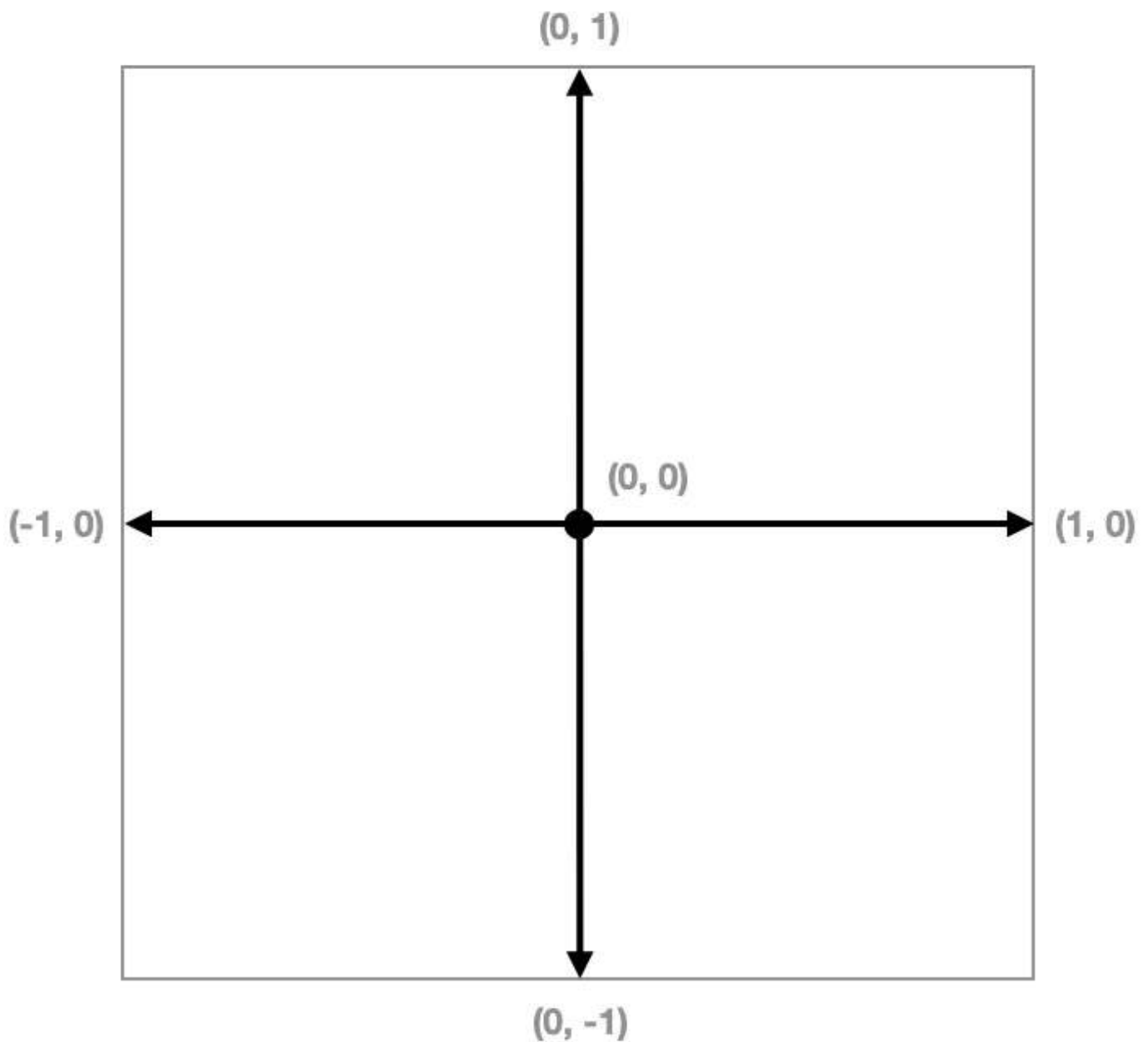
When we talk about positions, it is important to realize that positions are relative to some *coordinate system*.

A coordinate system consists of a point called the *origin* and a set of perpendicular unit-length *axes*. Given a coordinate system, a given point can be assigned a list of coordinates specifying how far away the point is from the origin along each axis. In 2D, we use  $x$  and  $y$  to denote these coordinates, while in 3D we add a  $z$  coordinate. So the point  $(1, 3, 2)$  is one unit (right) along the  $x$  axis, three units (up) along the  $y$  axis, and two units (toward us) along the  $z$  axis away from the origin. Because the origin is zero units away from itself, it is labeled  $(0, 0, 0)$ .

We will learn in future articles how to transform points from one coordinate system to another. For the time being, we won't worry too much about that. Just know that sometimes, it's easier to work in one system versus another, and there is a way to move between them.

## ***Normalized Device Coordinates***

In graphics, it is often useful to use “normalized” coordinate systems, where some significance is attached to the positions that are one unit away from the origin. One such space is *normalized device coordinate* (NDC) space, illustrated here:



What is the significance of the values -1 and 1 in NDC space? You can think of them as the boundaries of the picture we're drawing. For example, the point  $(1, 1)$  is at the top right of NDC, while the point  $(-1, -1)$  is at the bottom left. This is true regardless of the resolution (size) of the image, which is what makes NDC space convenient to work in.

The main purpose of a vertex function is to determine the position of each vertex. But in what space are these positions defined? For now, you can think of them as being in normalized device coordinates. (This is a lie, but it's one of those very useful lies.) This means that we'll be defining our shapes to draw with points whose x and y values are all between -1 and 1.

## ***A Renderer Class***

Now that we are starting to write more code in our sample apps, it is convenient to refactor some of the rendering code into its own class.



We define a `Renderer` class that holds the various Metal objects. Here is the part of the class definition that declares these members:

```
class Renderer: NSObject, MTKViewDelegate
{
    let device: MTLDevice
    let commandQueue: MTLCommandQueue
    let view: MTKView
    private var renderPipelineState: MTLRenderPipelineState!
    private var vertexBuffer: MTLBuffer!
```

By now, you know that we need a command queue to send commands to the GPU, and we need an `MTKView` to present our drawings to the screen. We also have a member of type `MTLRenderPipelineState`, which is a new pipeline state type we will introduce below.

To initialize a renderer, we provide it with a Metal device and a view to draw into. The renderer configures the view and assigns itself as the view's delegate so it knows when to draw.

```
init(device: MTLDevice, view: MTKView) {
    self.device = device
    self.commandQueue = device.makeCommandQueue()!
    self.view = view

    super.init()

    view.device = device
    view.delegate = self
    view.clearColor = MTLClearColor(red: 0.95,
                                     green: 0.95,
                                     blue: 0.95,
                                     alpha: 1.0)

    makePipeline()
    makeResources()
}
```

To see how this simplifies our view controller, here's the complete updated definition of the `ViewController` class:

```
class ViewController: NSViewController {
    @IBOutlet weak var mtkView: MTKView!
    var renderer: Renderer!

    override func viewDidLoad() {
```



```
        super.viewDidLoad()

        let device = MTLCreateSystemDefaultDevice()!
        renderer = Renderer(device: device, view: mtkView)
    }
}
```

We will return shortly to the `Renderer` class, but now we turn to the central topic of this article: render pipeline states.

## ***Render Pipeline States***

As we saw last time, we use pipeline state objects to tell our command encoder which shader function we want to run when executing subsequent commands. For example, we set the compute pipeline state containing our `add_two_values` kernel function when we wanted to add the values in two buffers, then dispatched a grid telling the GPU how many work items to execute.

When we want to encode drawing commands, we need to provide a render pipeline state. A render pipeline state encompasses a vertex function, a fragment function, and other values used to configure the GPU to our preferences. Any drawing commands (draw calls) we issue after setting the render pipeline state on the encoder will use that pipeline state's shaders to process the vertices and fragments of the draw call.

As with compute pipeline states, we create render pipeline states by requesting them from a device. However, because render pipelines are more complex than compute pipelines, we first fill out a *render pipeline descriptor*.

Render pipeline descriptors are an example of the parameter object pattern. They gather the various parameters needed to create a render pipeline state together, so they can be passed to the pipeline state creation method all at once.

You may have noticed a call to the renderer's `makePipeline()` method in the initializer above. This is where we will configure and create our pipeline state:

```
func makePipeline() {
    guard let library = device.makeDefaultLibrary() else
        { fatalError("Unable to create default Metal library")
        }

    let renderPipelineDescriptor = MTLRenderPipelineDescriptor()
    //...
```

First, we make sure we're able to get the app's default Metal shader library. Then we instantiate the render pipeline descriptor.

The vertex and fragment functions to run during the vertex and fragment stages of the pipeline are essential for doing anything useful, so we retrieve each function from the library and set it on the descriptor:

```
renderPipelineDescriptor.vertexFunction = library.makeFunction(name:
"vertex_main")!
renderPipelineDescriptor.fragmentFunction =
library.makeFunction(name: "fragment_main")!
```

There are many, many other possible variables we could set on the descriptor, but for now, the only other essential one is the color attachment's pixel format. This tells Metal the layout of the texture we will be drawing into. We set it to the color pixel format of the `MTKView`, since that is where our drawing will be happening.

```
renderPipelineDescriptor.colorAttachments[0].pixelFormat =
view.colorPixelFormat
```

Finally, we ask the device to create the render pipeline state by calling the `makeRenderPipelineState(descriptor:)` method. This operation can fail—for example, if the descriptor is invalid—so we wrap it in a `do...catch` block:

```
do {
    renderPipelineState = try
device.makeRenderPipelineState(descriptor: renderPipelineDescriptor)
} catch {
    fatalError("Error while creating render pipeline state: \
(error) ")
```



```
}  
}
```

## Preparing the Vertex Buffer

We're almost ready to start drawing, but first we need something to draw. Let's define a few points and write them into a buffer. We can define a new method called `makeResources()` in our renderer class to encapsulate this:

```
func makeResources() {  
    var positions = [  
        SIMD2<Float>(-0.8, 0.4),  
        SIMD2<Float>( 0.4, -0.8),  
        SIMD2<Float>( 0.8, 0.8)  
    ]  
    vertexBuffer = device.makeBuffer(bytes: &positions,  
                                    length:  
MemoryLayout<SIMD2<Float>>.stride * positions.count,  
                                    options: .storageModeShared)  
}
```

We use a different buffer creation method called

`makeBuffer(bytes:length:options)` this time, since we have created the list of vertex positions in advance. This creates the buffer *and* copies the positions into it in one step. We could also have used the `makeBuffer(length:options:)` method as we did before, but then we'd have to copy the points in separately.

Note that the x and y coordinates of each point are between -1 and 1. This means that once they pass through the vertex function, they will be in NDC space already. If you can't quite visualize where they are, consider drawing a graph and plotting them, then noticing that they can be joined into a large triangle.

## Encoding Draw Calls

Since we have made our renderer the delegate of our `MTKView`, its `draw(in:)` method will be called each frame to update the view's contents.

As we did when clearing the view on Day 4, we start our draw method by



asking the view for its current render pass descriptor, then making a command buffer:

```
func draw(in view: MTKView) {
    guard let renderPassDescriptor =
view.currentRenderPassDescriptor else { return }

    guard let commandBuffer = commandQueue.makeCommandBuffer() else
{ return }
    //...
```

We know we'll be issuing render commands, so we create a render command encoder from the pass descriptor:

```
let renderCommandEncoder =
commandBuffer.makeRenderCommandEncoder(descriptor:
renderPassDescriptor) !
```

When we want to draw something, we do it in three steps:

1. Set any state we want on the render command encoder, including the render pipeline state object.
2. Set any resources we want to use in our draw calls; in this case, that's just the buffer containing the vertex positions
3. Encode draw calls, describing the types of primitive to draw and the number of vertices to use.

Below, each of these steps is executed in turn. We'll be drawing one triangle, so we specify `.triangle` as the primitive type and `3` as the vertex count:

```
renderCommandEncoder.setRenderPipelineState(renderPipelineState)
renderCommandEncoder.setVertexBuffer(vertexBuffer,
                                     offset: 0,
                                     index: 0)
renderCommandEncoder.drawPrimitives(type: .triangle,
                                     vertexStart: 0,
                                     vertexCount: 3)
```



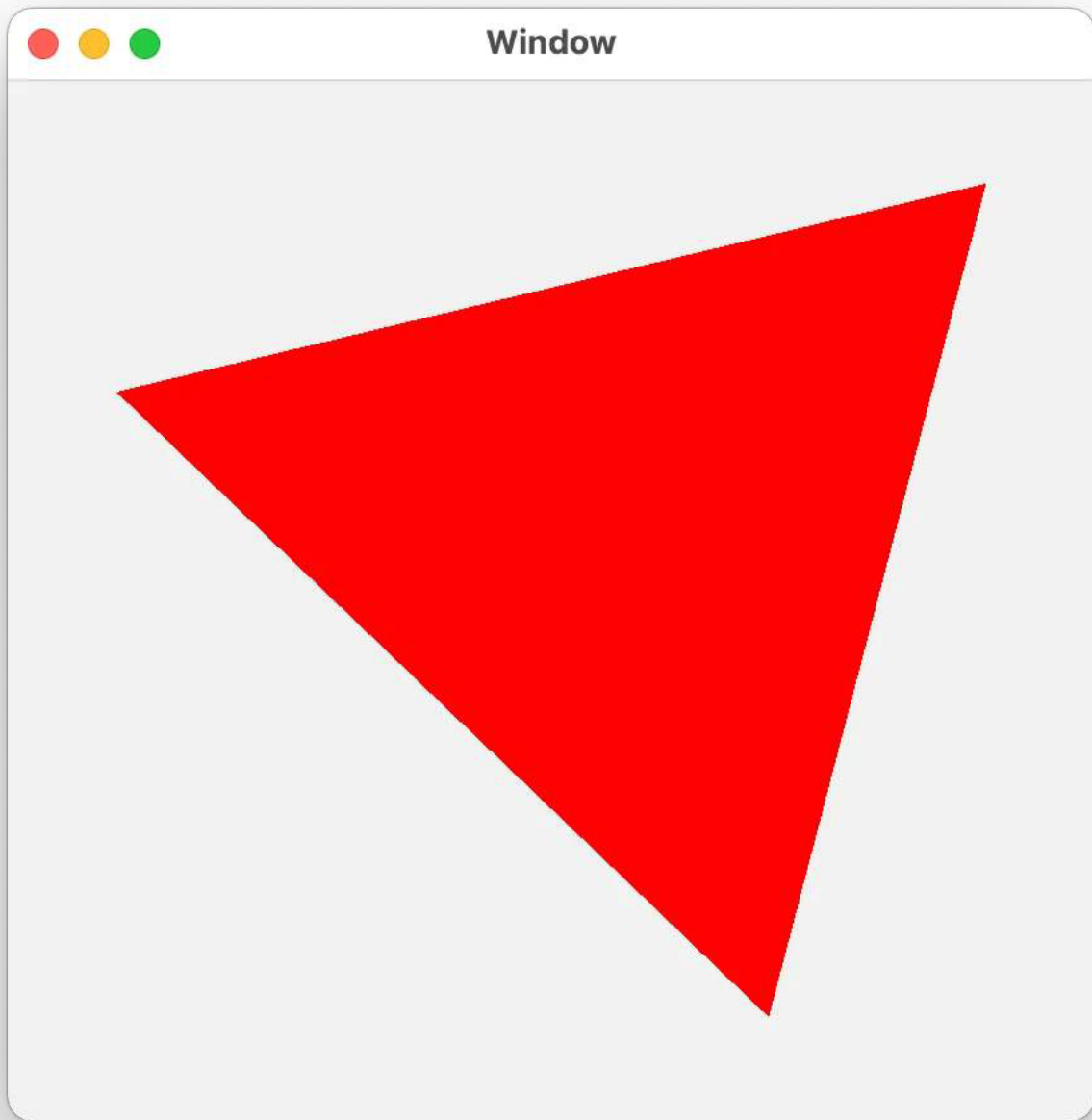


We can issue as many draw calls as we want in each encoder, even switching between render pipeline states between them if we want.

For now, we're just trying to get our first triangle on the screen, so we do our usual work to end the frame: end encoding, present the drawable, and commit the command buffer.

```
renderCommandEncoder.endEncoding()  
  
commandBuffer.present(view.currentDrawable!)  
commandBuffer.commit()  
}
```

If all has gone according to plan, we can build and run to see the results of our labors: the first triangle, with many more to come.



After learning a lot of concepts and writing a lot of code, we finally achieved our first milestone: drawing a triangle on the screen. In the next article, we'll talk about how to extend the amount of data processed by our pipeline by adding more attributes to our vertices. Stay tuned!

# Day 8: Vertex Attributes



Warren Moore ·

5 min read · Apr 9, 2022

*This series of posts is my attempt to present the Metal graphics programming framework in small, bite-sized chunks for Swift app developers who haven't done GPU programming before.*

*If you want to work through this series in order, start [here](#).*

In the previous article, we crossed the threshold from learning foundational concepts of Metal into actually drawing shapes with the GPU. In this article, we will augment the previous example by adding a new *attribute* to our vertices — color — and learn how to simplify our vertex function signatures by using *vertex descriptors*.

## **Attributes: Beyond Vertex Positions**

When writing our first vertex function, we took two parameters: a pointer to a buffer containing two-element float vectors (positions), and a vertex ID:

```
vertex float4 vertex_main(  
    device float2 const* positions [[buffer(0)]],  
    uint vertexID [[vertex_id]])
```

By indexing into the buffer, we could retrieve the position of the vertex and operate on it.

But what if we wanted our vertex to carry more data than just its position? When we want to render realistic lighting, we need to know which way the surface is oriented at each vertex, which is provided as a vector called the *vertex normal*. We might also want to apply a texture map to a surface,

which requires us to include *texture coordinates*.

Each of these vertex properties — position, normal, texture coordinates — is called an *attribute*, and vertices can have many attributes.

We can gather the attributes of each vertex into a struct in our shader code, to make things easier to manage. Here's a vertex struct with a position attribute and a color attribute:

```
struct VertexIn
{ float2 position;
  float4 color;
};
```

With the attributes of the vertex packaged together, we could interleave positions and colors in our vertex buffer, and update our vertex function signature to take a pointer to these structs:

```
vertex VertexOut vertex_main(
    device VertexIn const* vertices [[buffer(0)]],
    uint vertexID [[vertex_id]])
{
    VertexIn in = vertices[vertexID];
```

The return type `VertexOut` is another structure that gathers the output attributes of the vertex function, so they can be interpolated by the rasterizer:

```
struct VertexOut {
    float4 position [[position]];
    float4 color;
};
```

## ***Alignment Considerations***

Back in our Swift code, we might imagine that we could update our vertex buffer by just adding color information to the buffer:



```
func makeResources() {
    var vertexData: [Float] = [
        //      x      y      r      g      b      a
        -0.8,  0.4,    1.0, 0.0, 1.0, 1.0,
    ]
    vert 0.4, -0.8,    0.0, 1.0, 1.0, 1.0,
        0.8,  0.8,    1.0, 1.0, 0.0, 1.0,
        length: MemoryLayout<Float>.stride * vertexData.count,
        options: .storageModeShared)
}
```

Superficially, this looks like it should work, but we've overlooked something crucial: alignment. The `float4` vector type has an alignment of 16 bytes, meaning that Metal expects every `float4` to start at a 16-byte boundary. But `float2` only has an alignment of 8 bytes, so the color values we wrote into our buffer are *misaligned* and will not be loaded correctly.

We could alleviate this by padding the array of input values with extra floating point values to fill up the space between our positions and colors:

```
func makeResources() {
    var vertexData: [Float] = [
        //      x      y (pad pad) r      g      b      a
        -0.8,  0.4, 0.0, 0.0, 1.0, 0.0, 1.0, 1.0,
        0.4, -0.8, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0,
        0.8,  0.8, 0.0, 0.0, 1.0, 1.0, 0.0, 1.0,
    ]
    vertexBuffer = device.makeBuffer(
        bytes: &vertexData,
        length: MemoryLayout<Float>.stride * vertexData.count,
        options: .storageModeShared)
}
```

But there's a better way: vertex descriptors.

## Vertex Descriptors

What we want is a way to decouple how we lay out our vertex data in our buffers from how we lay out our vertex structs in our shaders.

A *vertex descriptor* is an object that tells Metal how vertex attributes are arranged in

vertex buffers. It consists of an array of *attributes* and an array



of buffer *layouts*.

We create a vertex descriptor by just instantiating it:

```
let vertexDescriptor = MTLVertexDescriptor()
```

We specify each vertex attribute by providing its format, its offset, and its buffer index. The *format* indicates the type of the attribute in the buffer (`float2`, `float4`, etc.). The *offset* is the number of bytes from the start of the vertex to the attribute data. The *buffer index* indicates which vertex buffer holds the data.

For example, here is how we might describe our position and color attributes:

```
vertexDescriptor.attributes[0].format = .float2
vertexDescriptor.attributes[0].offset = 0
vertexDescriptor.attributes[0].bufferIndex = 0

vertexDescriptor.attributes[1].format = .float4
vertexDescriptor.attributes[1].offset = MemoryLayout<Float>.stride *
2
vertexDescriptor.attributes[1].bufferIndex = 0
```

Position is the first attribute, and each position is a two-element float vector (`.float2`). Since positions come first in the buffer, their offset is `0`. Since we're interleaving positions and colors in the same buffer (buffer 0), we set the buffer index to 0.

Color is the second attribute; it has a format of `.float4`, since it contains 4 components (RGBA). It is offset 8 bytes (two floats' worth of space) from the start of the vertex, and occupies the same buffer as our positions (buffer 0).

The only other bit of information we need to provide is the *stride* between vertices, the total number of bytes a single vertex occupies.

Since all of our vertex data is in one buffer, we only need to configure the

first buffer layout's stride:

```
vertexDescriptor.layouts[0].stride = MemoryLayout<Float>.stride * 6
```

This informs Metal that each vertex occupies the space of 6 floats, or 24 bytes.

Metal needs to know our vertex attributes and layout at shader compile time, so when building our pipeline(s), we set the `vertexDescriptor` property of our render pipeline descriptor to the vertex descriptor we just configured:

```
renderPipelineDescriptor.vertexDescriptor = vertexDescriptor
```

## ***Adapting Shaders to Use Vertex Descriptors***

Now that we are using a vertex descriptor, the job of writing vertex shaders gets easier. We just need to tell Metal which vertex struct members match which attributes, and Metal will automatically generate code to fetch vertex data from our buffer(s):

```
struct VertexIn {  
    float2 position [[attribute(0)]];  
    float4 color      [[attribute(1)]];  
};
```

We don't need to worry about alignment anymore, since each struct member will be populated according to the information we provided in the vertex descriptor.

We can now simplify our vertex function signature as well:

```
vertex VertexOut vertex_main(VertexIn in [[stage_in]])
```

`stage_in` indicates that we expect Metal to fetch vertices on our behalf, so we can eliminate the `vertex_id` parameter.

The rest of the vertex function continues much as it previously did, but now we also pass the vertex color through to the output:

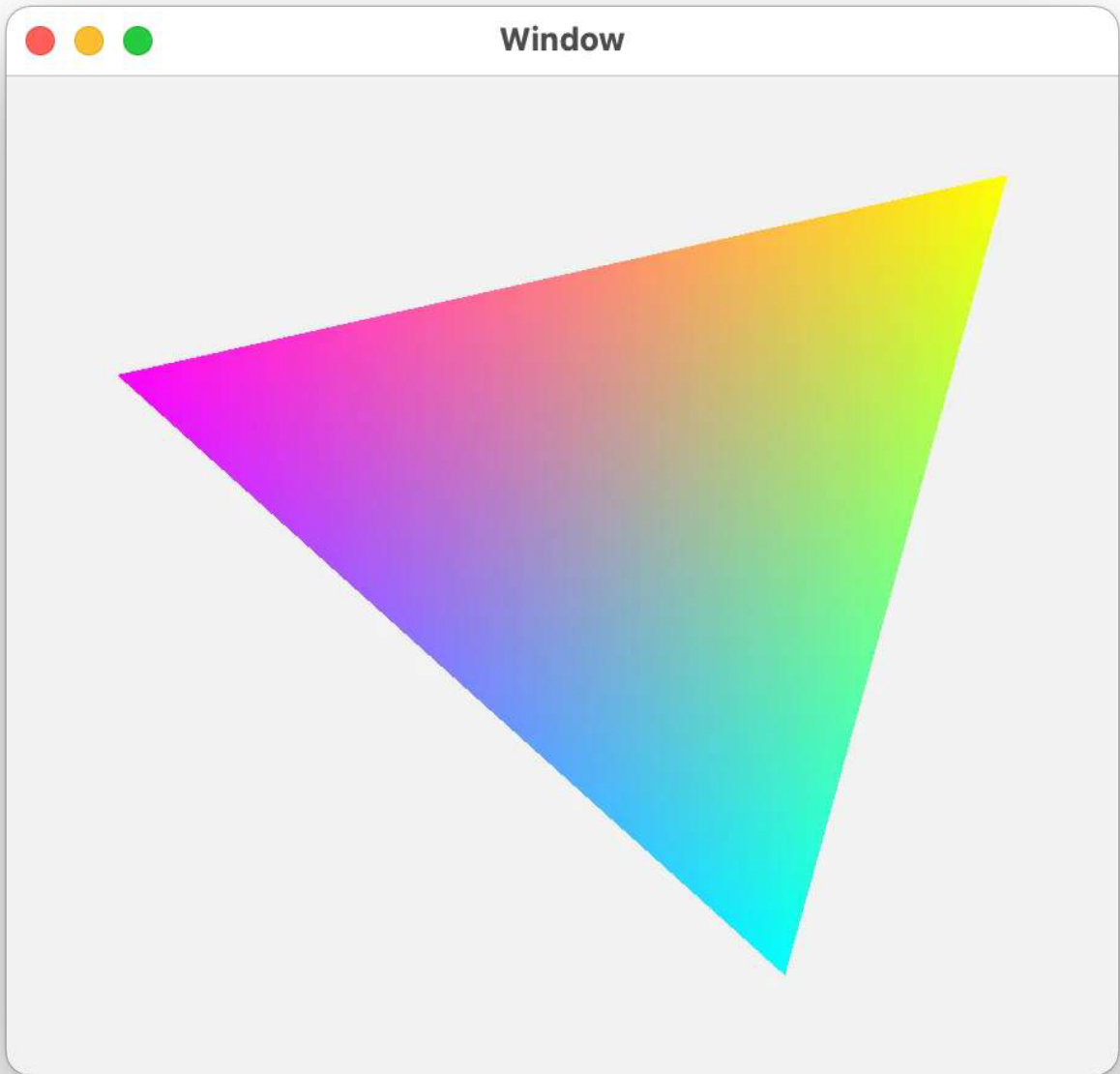
```
vertex VertexOut vertex_main(VertexIn in [[stage_in]])
{
    VertexOut out;
    out.position = float4(in.position, 0.0, 1.0);
    out.color = in.color;
    return out;
}
```

Tidy!

Since our vertices now have a color attribute, the rasterizer will automatically generate a color for each fragment by interpolating among the vertices. We can visualize this by returning the fragment color from the fragment function:

```
fragment float4 fragment_main(VertexOut in [[stage_in]])
{ return in.color;
}
```

Running the app produces a much more colorful triangle than before:



That's how we add additional attributes to vertices and use Metal to simplify vertex shader authoring with vertex descriptors. Next time, we'll look at how to feed other data besides vertex attributes into our shader functions, which will give us a glimpse of how to introduce interaction and animation into our Metal apps.

# Day 9: Constants



Warren Moore ·

6 min read · Apr 10, 2022

*This series of posts is my attempt to present the Metal graphics programming framework in small, bite-sized chunks for Swift app developers who haven't done GPU programming before.*

*If you want to work through this series in order, start [here](#).*

Last time, we talked about how to add attributes to vertices so we could fill our triangle with smoothly shaded colors. In this article we will talk about how to pass constant data to shaders, which remains the same for all vertices in a draw call.

## Constant Data

We call this data *constant data* because it doesn't change between invocations of the shader function. This contrasts with both attribute data, which can change per-vertex, and interpolated data, which can change per-fragment. Some APIs (notably OpenGL) call these types of constant values *uniforms*, while most others use the term *constants*.

Like vertices, constant shader parameters can be ordinary types like `float`, `float2`, etc.; or they can be structs. It is often useful to collect together the constants that change at the same “frequency”—per frame, per draw call, etc.—so as to minimize the number of times you need to set buffers on the command encoder.

## Getting Constants into Shaders

Suppose we want to animate the triangle we have been drawing in the past several articles. One of the simplest things we could do is provide a vector

that we use to offset the position of each vertex. By updating this vector over time, we could make the triangle move around the screen.

We can achieve this effect by using a dynamic constant in our vertex shader. The phrase “dynamic constant” might seem like a contradiction in terms, but it really just refers to constants that change over time and less often than once per draw call.

We modify the vertex function to take a constant reference to `float2` and indicate that it will be available in buffer 1, since buffer 0 holds our vertex data:

```
vertex VertexOut vertex_main(  
    VertexIn in [[stage_in]],  
    constant float2 &positionOffset [[buffer(1)]])
```

We update the body of the vertex function to add this offset vector to the current vertex’s position, moving by the specified amount in NDC space:

```
out.position = float4(in.position + positionOffset, 0.0, 1.0);
```

To get the constant data into the shader, we use the exact same method to bind the constant buffer that we used to bind the vertex buffer. We bind it to buffer slot 1, corresponding to the `buffer(1)` in the shader code:

```
renderCommandEncoder.setVertexBuffer( co  
    nstantBuffer,  
    offset: currentConstantBufferOffset,  
    index: 1)
```

Notice that we supply an offset when binding the buffer, which we haven’t done before. To understand why, we need to talk a little about multiple buffering and data synchronization.

## Triple Buffering

Recall that the CPU and GPU work in parallel: the GPU might be running previously-encoded commands while we're encoding the next frame on the CPU. For this reason, we need to ensure that we don't change data out from under the GPU, which could cause corruption or even a crash.

By default, a Metal layer has three drawables available. Since we need a drawable (and the texture it contains) in order to encode a frame, say that a view can have up to three frames “in flight” at a time.

We declare a global constant to indicate that we allow up to three frames to be processing at once (this is called “triple buffering”):

```
let MaxOutstandingFrameCount = 3
```

To help keep track of which frame we're rendering, we'll add a frame count member to the renderer class, which we'll increment once we're done encoding each frame:

```
private var frameIndex: Int
```

Since we can have three frames in flight, we also allocate three memory regions for our constants, to ensure that the constants we wrote for a previous frame don't get overwritten before the GPU is done with them.

We don't need to allocate three separate buffers to hold our per-frame constants. Instead, we can allocate a buffer that is three times the necessary size, and use a per-frame offset to determine the correct region to write into.

We'll add several members to our renderer class to keep track of the constant buffer, the size and stride of the constant data, and the offset into the constant buffer that corresponds to the current frame:

```
private var constantsBuffer: MTLBuffer!
private let constantsSize: Int
private let constantsStride: Int
private var constantsBufferOffset: Int
```

We set up these values in our initializer:

```
self.frameIndex = 0
self.constantsSize = MemoryLayout<SIMD2<Float>>.size
self.constantsStride = align(constantsSize, upTo: 256)
self.constantsBufferOffset = 0
```

Note that we store the size and stride of the constants separately. Even though the constants themselves are quite small (only 8 bytes in total), some GPUs have a limit on how granular the buffer offset can be. Therefore, in this case, the stride is 256 bytes per constant vector.

Now that we know how big our constant buffer needs to be, we allocate it in the

`makeResources()` method:

```
constantBuffer = device.makeBuffer(
    length: constantsStride * MaxOutstandingFrameCount,
    options: .storageModeShared)
```

## ***Data Synchronization***

We know we have enough space to store three frames' worth of constants, but how to we ensure that we access each region at the right time? We need some kind of synchronization primitive that lets us encode up to three frames, but then waits until one completes before beginning the next.

It turns out that the Dispatch framework has just the thing:

`DispatchSemaphore`, which is a “counting semaphore” implementation.

We initialize a Dispatch semaphore with a value that will be decremented at the start of each frame and incremented at the end of each frame. Any time





its value is zero, asking it to decrement will cause it to first block the current thread until it is incremented on another thread.

```
private var frameSemaphore = DispatchSemaphore(value:
    MaxOutstandingFrameCount)
```

We structure our draw method around this semaphore. Upon entry to the `draw(in:)` method, we wait on the semaphore, then we encode our rendering work, then we add a completed handler to the command buffer that will increment the semaphore (by calling `signal()`). Before returning we increment the frame count.

```
func draw(in view: MTKView) {
    // This blocks if three frames are already underway
    frameSemaphore.wait()

    updateConstants()
    // ... encode work ...

    commandBuffer.addCompletedHandler { [weak self] _ in

        // This unblocks the waiting thread, if any
        self?.frameSemaphore.signal()
    }
    frameIndex += 1
}
```

We already saw above how to bind the constant buffer prior to issuing our draw call; everything else about our render command encoding is the same.

## ***Updating Per-Frame Constants***

Updating the constants consists of calculating the position offset vector from the current time, then writing the vector into the constant buffer at the current offset. We add an `updateConstants()` method to do this work.

First, the offset vector calculation. We take the current time, turn it into a rotation angle by multiplying it by a speed factor and taking the result mod  $2\pi$ . Then we find the vector by taking the cosine and size of the angle as our x and y coordinates, respectively:

```
func updateConstants() {
    let time = CACurrentMediaTime()
    let speedFactor = 3.0
    let rotationAngle = Float(fmod(speedFactor * time, .pi * 2))
    let rotationMagnitude: Float = 0.1
    var positionOffset =
        rotationMagnitude * SIMD2(cos(rotationAngle),
                                           sin(rotationAngle))

    // ...
}
```

To determine where we will write the vector, we find the constant buffer offset. It is equal to the current frame index modulo the maximum frame index (3) multiplied by the constants stride:

```
constantsBufferOffset =
    (frameIndex % MaxOutstandingFrameCount) * constantsStride
```

The expression `frameIndex % MaxOutstandingFrameCount` cycles through the sequence 0, 1, 2, 0, 1, 2, etc., so after we write the constants for the third frame, we reuse the region of the buffer that was used for the first frame. This type of structure is called a *circular buffer* or *ring buffer*.

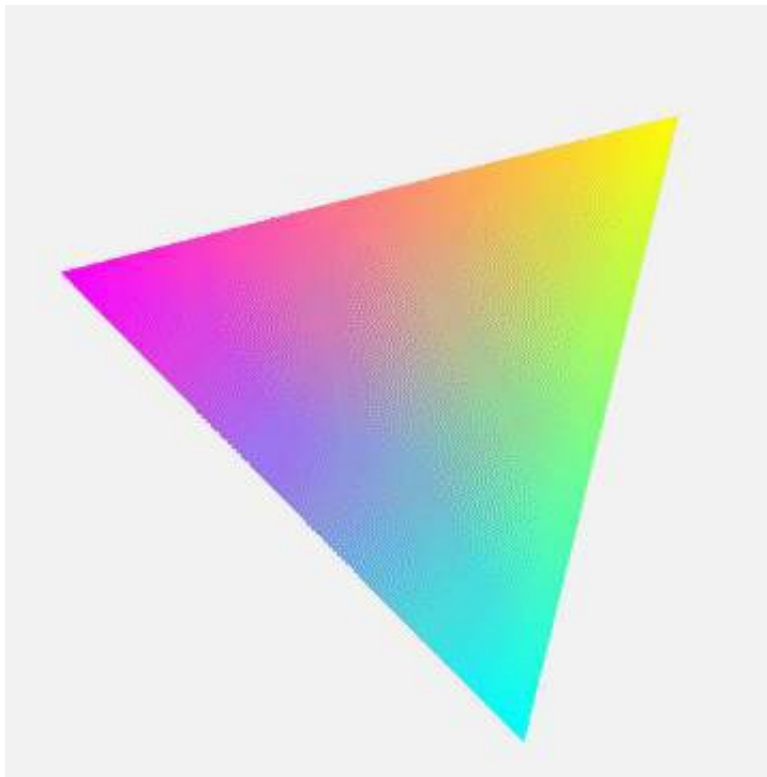
To get a pointer to the current constant buffer region, we use the

`advanced(by:)` method on the `contents()` pointer of the buffer. We then copy the position offset vector into the buffer with the `copyMemory(from:byteCount:)` method.

```
let constants = constantsBuffer.contents()
    .advanced(by: constantsBufferOffset)

constants.copyMemory(from: &positionOffset,
                    byteCount: constantsSize)
}
```

If we build and run our updated app, we can see the triangle gently circling the screen:



Next time, we'll look at some 2D math that will help us leverage the power of animation and interaction in the future.

# Day 10: 2D Math



Warren Moore ·

15 min read · Apr 11, 2022

*This series of posts is my attempt to present the Metal graphics programming framework in small, bite-sized chunks for Swift app developers who haven't done GPU programming before.*

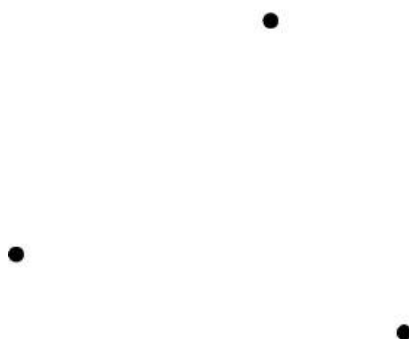
*If you want to work through this series in order, start [here](#).*

## Points and Vectors

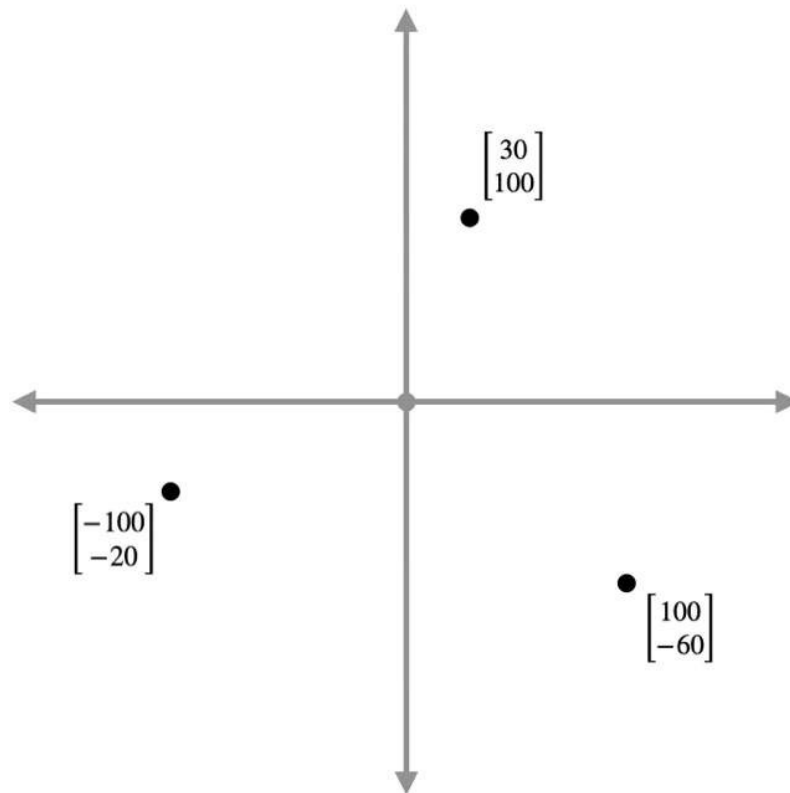
So far, we've made the simplifying assumption that all vertex positions are expressed in normalized device coordinates (NDC), which conveniently happens to match what Metal expects from the vertex shader.

If we eventually want to do useful work in three dimensions, though, we need to understand how to move between different coordinate systems. We'll start by distinguishing points from vectors and discovering the basic operations we can perform on them.

Suppose we have some points.

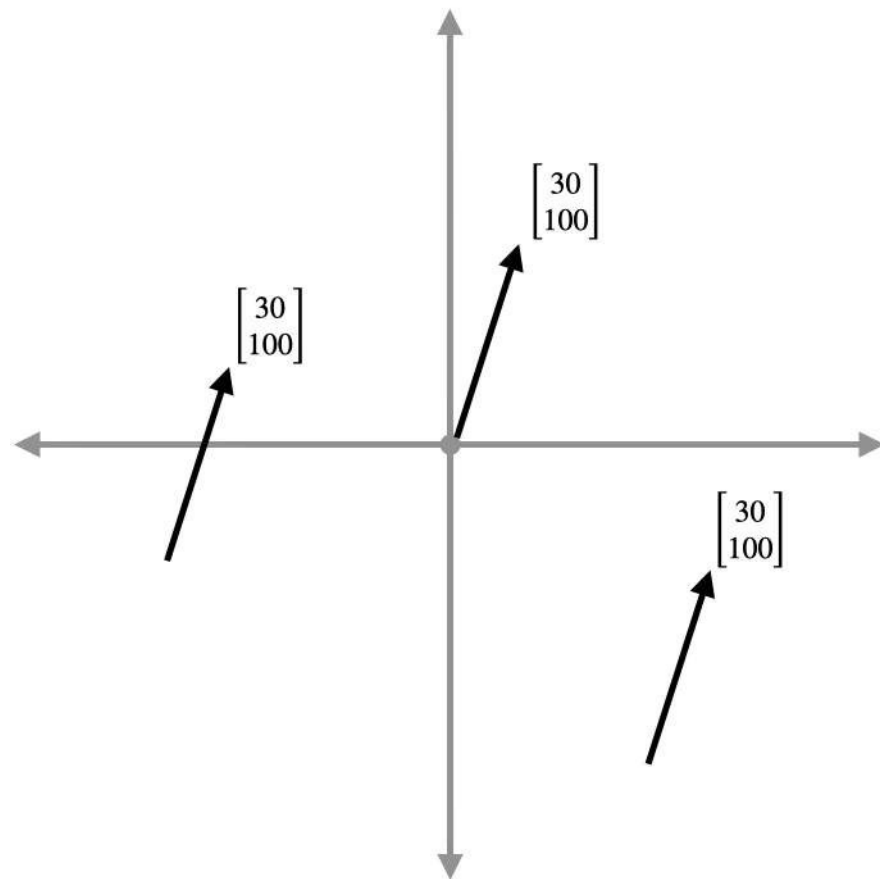


Okay, that's not especially useful. We don't know where these points are, or how far apart they are, because we haven't established a coordinate system. Let's do that and give them some labels.



Much better. Now we can see that the origin of the coordinate system is in the middle of the space, and the points are each about one hundred units away from the origin.

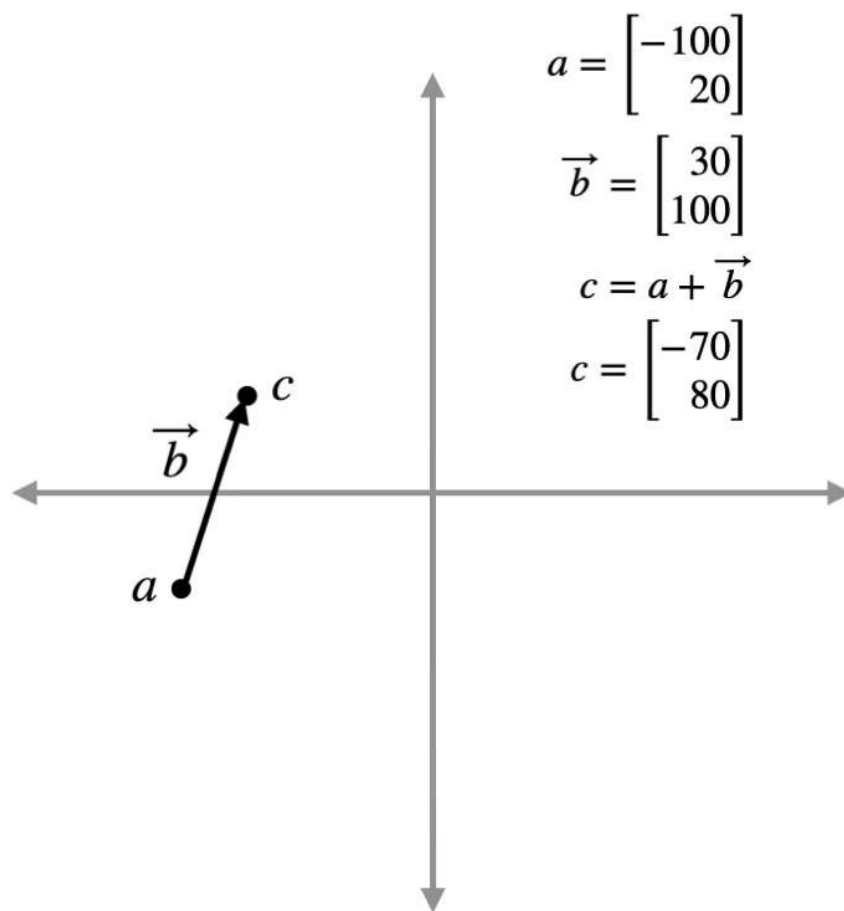
In addition to points, we need to get familiar with vectors. Whereas a point is a *location* in a space, a vector is a *displacement* in a space. This means that a vector's coordinates are the same regardless of “where” it is in space.



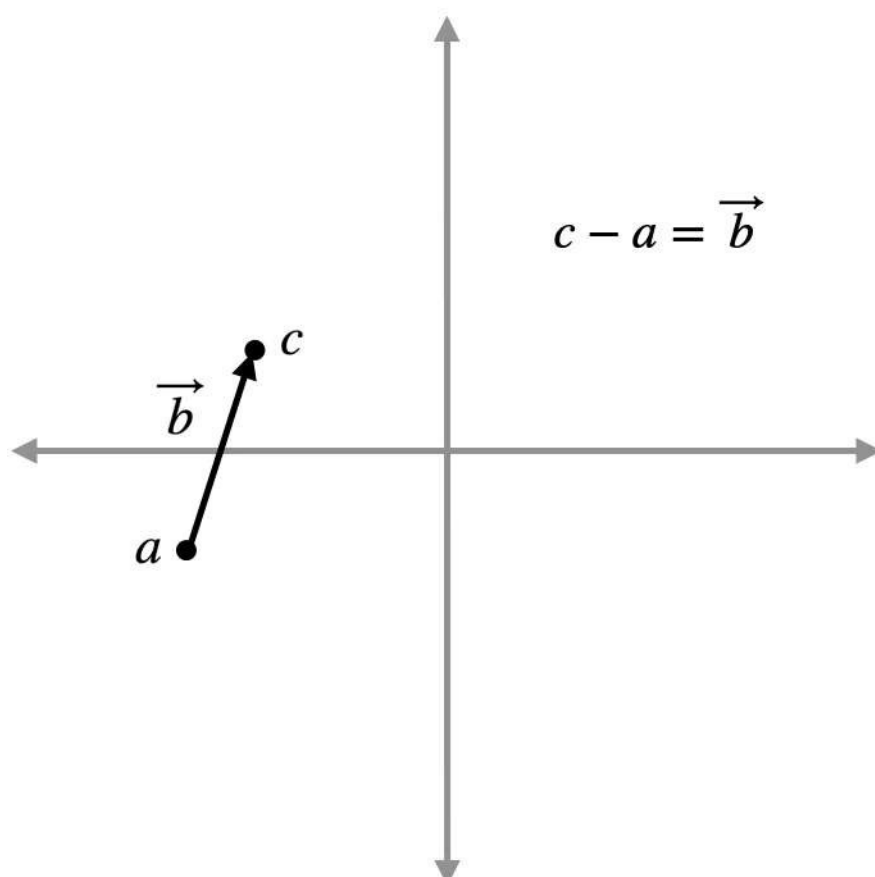
## ***Point and Vector Arithmetic***

This point-vector distinction is crucial, because it allows us to define sensible arithmetic operations between points and vectors. For example, we can move from one point to another by adding a vector to the initial point.

Here, the point `a` is displaced by the vector `b` to arrive at the point `c`.



It is also meaningful to subtract one point from another. The result is a vector that points from the first point to the second point. This follows immediately from rearranging the equations above.



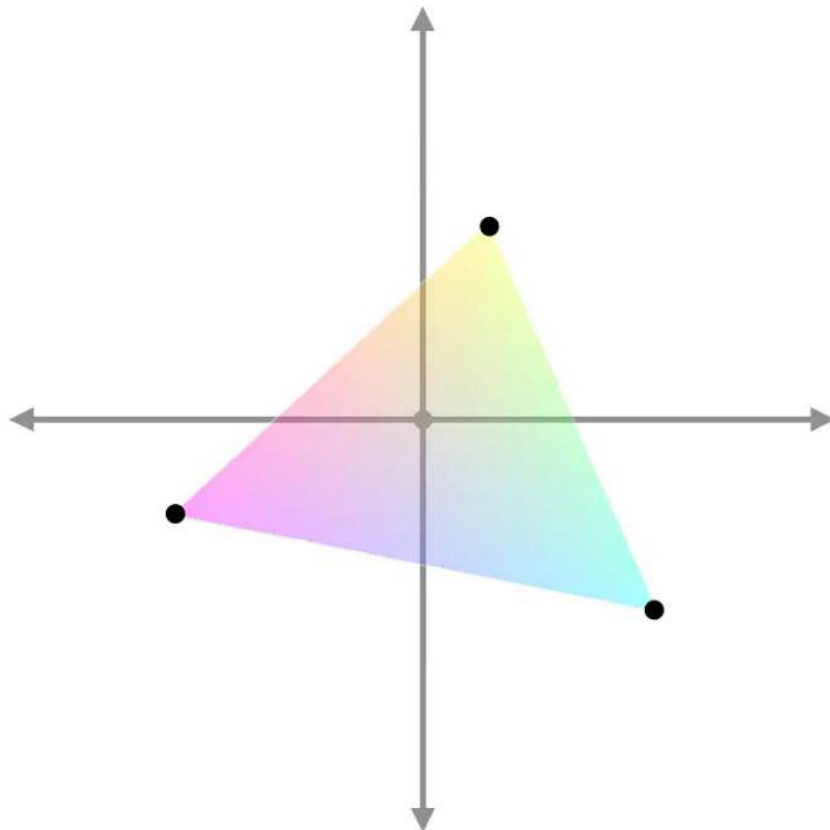


Adding two vectors also makes sense. We do this geometrically by placing the “tail” of the second vector at the “tip” of the first vector. The displacement from the tail of the first vector to the tip of the second vector is then the displacement of their sum, which is yet another vector.

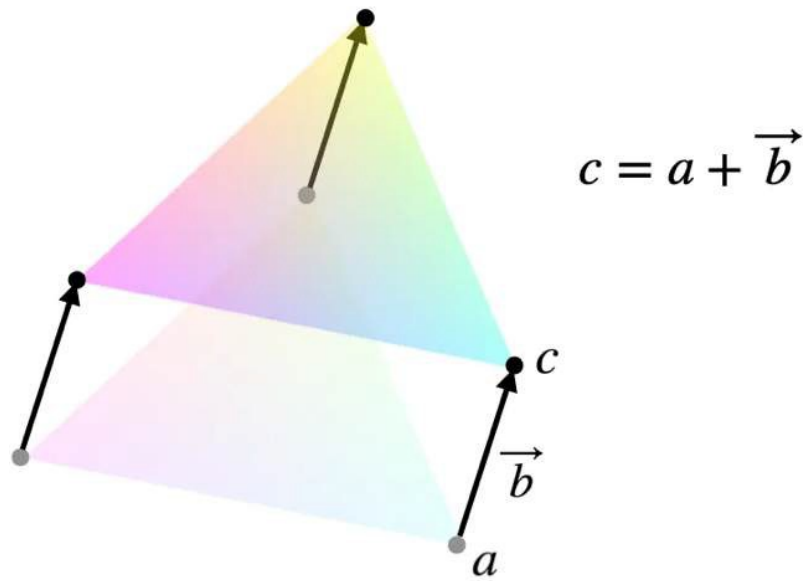
We can “convert” from a vector to a point by adding the vector to the origin. Since we already know that adding a vector to a point is valid, and the origin is just a point, we expect their sum to be a point that is displaced from the origin according to the vector.

## ***Translating Points***

Let’s connect our points together into a triangle so we can see how various operations affect our shapes.



Adding a constant vector to each point of a shape moves the shape in space. This is called *translation*.



In the figure above, we've removed the origin, axes, and labels to emphasize the geometric nature of translation, but if we actually wanted to perform the translation in code (as we will later), we'd have to write down the points and vector in a particular coordinate system, then carry out the addition.

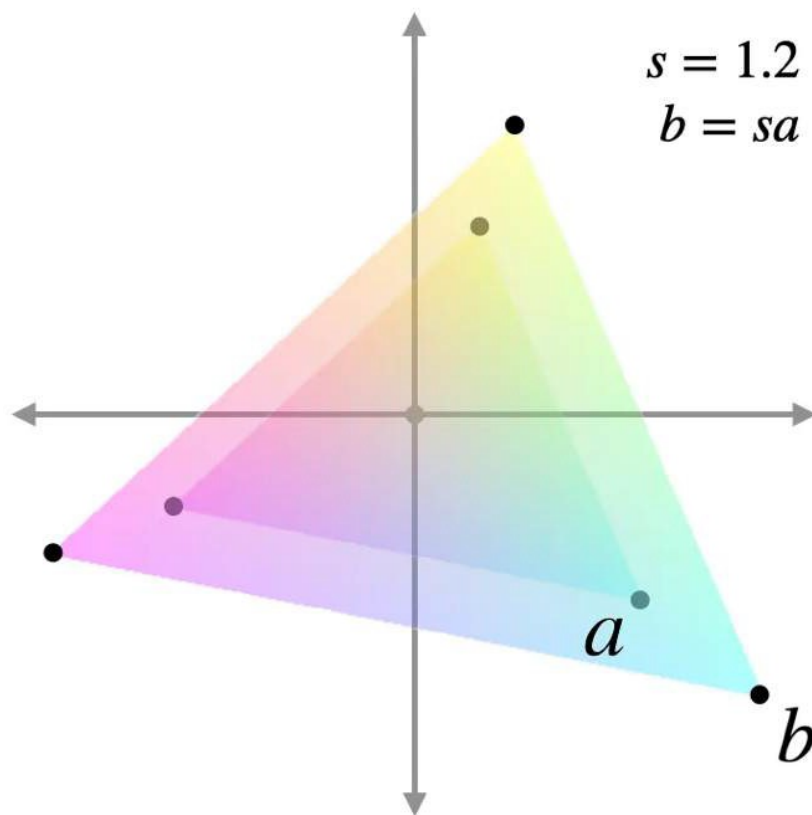
## Scaling Points

We have looked at several examples of addition and subtraction among points and vectors, but what happens when we multiply a point by a scalar (a single number)?

We can gain some intuition for this by considering how numbers on a one-dimensional number line move when multiplied: a multiplicative factor of greater than one causes points on a number line to move away from the origin (proportionately to their distance from the origin), and a multiplicative factor of less than one causes points to move toward the origin.

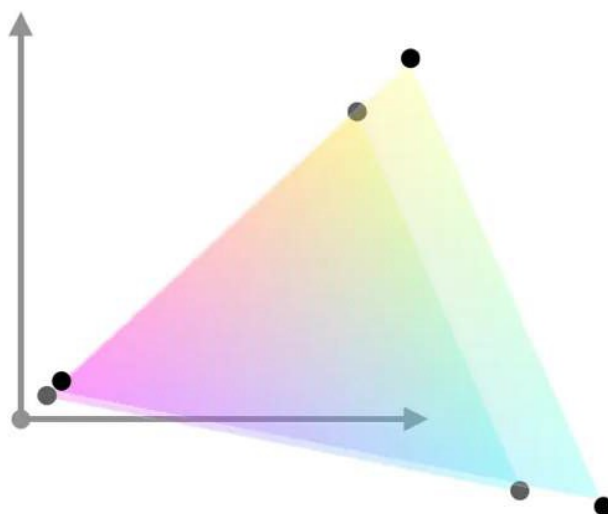
Likewise, if we multiply *points* by a scalar, they move toward or away from the origin. We call this process *scaling*.

If we consider our friendly triangle again and imagine multiplying its points by some factor like 1.2, we can observe scaling in action.



The vertices of the triangle are still roughly uniformly distributed about the origin because the center of the triangle nearly coincides with the origin.

If we scale a figure that is *not* centered around the origin, the points still scale proportionately, but rather than appearing to scale about the origin, they seem to scale away from it.



This illustrates that the coordinate space we model our shapes in matters. For example, if we were designing a bipedal character, we might place the origin of its local coordinate system between its feet, so that when it's scaled



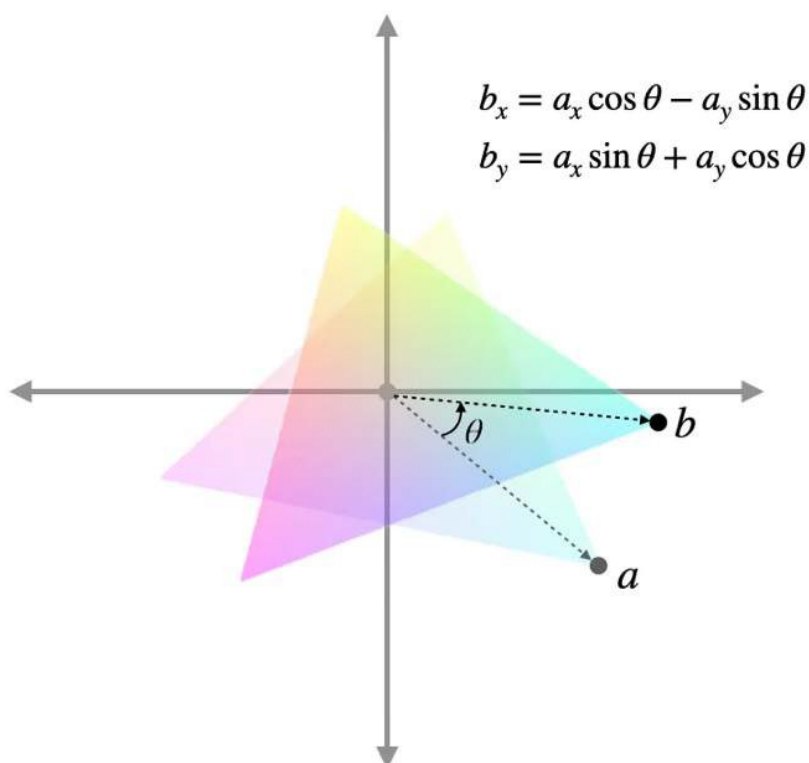
up, it would scale away from the ground plane while its feet remain planted. Other times, it's more convenient to place the origin at the centroid, or average, of the points. It all depends on what makes it easiest to compose objects together in a larger virtual world.

## Rotating Points

Translation and scaling are two examples of *transformations*. A transformation is a way of moving or reshaping a figure.

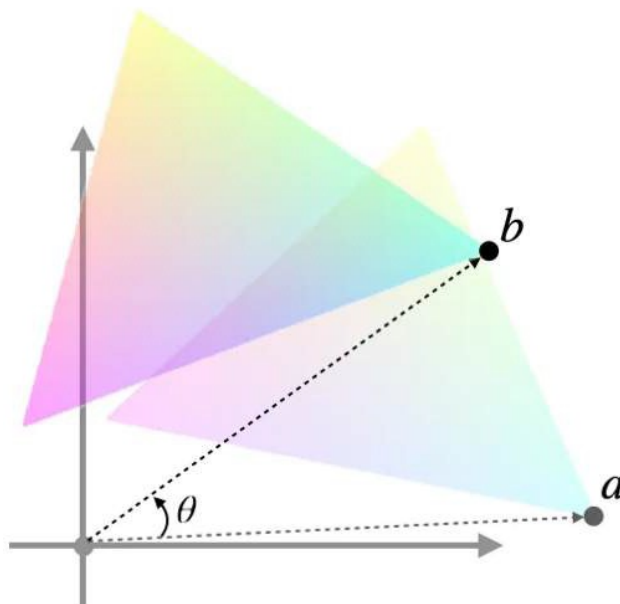
The last fundamental transformation we will talk about is rotation.

In two dimensions, there is only one axis that we can rotate around: the z axis. The z axis is perpendicular to both the x and y axes; it projects “out of the screen” toward us. Rotating around the z axis by a particular angle makes points orbit around the origin.



Rotation can seem intimidating at first because it involves trigonometric functions, but if this is your first time seeing these formulas, you shouldn't worry about memorizing or understanding them. We will package up the math into utility functions that just do the right thing. We're developing a vocabulary of transformations here, not taking a math class.

Rotating a figure that is not centered on the local origin has the effect of making the entire object move around the origin rather than its own centroid. This is often what we want. Think of a camera orbiting around a scene in “bullet time”: the camera’s path through space is an arc focused on the center of the action.



## Composing Transformations

We have talked about three fundamental types of transformations: translation, rotation, and scaling. The formulas for these transformations each look quite distinct. Translation is just vector addition, scaling is multiplying by a scalar, and rotation is a linear combination of vector components multiplied by trigonometric functions.

Suppose we want to chain different transformations together. One extremely common sequence of transformations is: scale, then rotate, then translate. We could write down an equation that does all three like this:

$$b_x = sa_x \cos \theta - sa_y \sin \theta + \vec{t}_x$$

$$b_y = sa_x \sin \theta + sa_y \cos \theta + \vec{t}_y$$

This certainly would work. But sometimes we want to chain together arbitrary transformations in arbitrary orders. It would be nice not to have to write different code every time we want a different sequence.

Perhaps there's some way to write down all of these different transformations as the same kind of thing...

## ***Enter the Matrix***

It turns out that we can write scaling, rotation, translation, and many other kinds of transformation as matrices.

First of all, though, what *is* a matrix?

Just as a scalar is a single number and a vector is a one-dimensional list of numbers, a *matrix* is a two-dimensional array of numbers. That's it. All of the power of matrices comes from the mathematical rules we apply to them.

A matrix with N rows and M columns is called an N×M matrix. We identify an individual element in a matrix with a subscript pair that indicates its row and column. For example, in the matrix **A** below, element **a<sub>02</sub>** is in the first row (row 0) and the third column (column 2).

$$A = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix}$$

## ***Matrix Multiplication***

It turns out that the only rule we need to unify transformations together in the language of matrices is *matrix multiplication*. Matrix multiplication isn't quite as simple as scalar or vector multiplication, but it follows simple rules.

Suppose we want to multiply two matrices,  $B$  and  $A$ , forming a matrix product,  $BA$ . For two matrices to be compatible, the number of columns in the left-hand matrix must be equal to the number of rows in the right-hand matrix.  $B$  and  $A$  are both  $3 \times 3$  matrices, so they are compatible with this rule.

Each element in the product matrix is found by taking the corresponding row from the left matrix and the corresponding column from the right matrix, multiplying them together pairwise, and summing up the results.

As an example, to find the element in the second row and third column of the product  $BA$ , we take the second row of  $B$  and the third column of  $A$ , multiply each pair of their respective elements, and add up the products.

$$\begin{aligned}
 B &= \begin{bmatrix} b_{00} & b_{01} & b_{02} \\ b_{10} & b_{11} & b_{12} \\ b_{20} & b_{21} & b_{22} \end{bmatrix} \\
 A &= \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix} \\
 BA &= \begin{bmatrix} b_{00}a_{00} + b_{01}a_{10} + b_{02}a_{20} & b_{00}a_{01} + b_{01}a_{11} + b_{02}a_{21} & b_{00}a_{02} + b_{01}a_{12} + b_{02}a_{22} \\ b_{10}a_{00} + b_{11}a_{10} + b_{12}a_{20} & b_{10}a_{01} + b_{11}a_{11} + b_{12}a_{21} & b_{10}a_{02} + b_{11}a_{12} + b_{12}a_{22} \\ b_{20}a_{00} + b_{21}a_{10} + b_{22}a_{20} & b_{20}a_{01} + b_{21}a_{11} + b_{22}a_{21} & b_{20}a_{02} + b_{21}a_{12} + b_{22}a_{22} \end{bmatrix}
 \end{aligned}$$

Carrying out this procedure for every element gives the complete matrix product.

## Expressing Transformations as Matrices

For reasons that are out of the scope of this article, it takes an  $(N + 1) \times (N + 1)$ -dimensional matrix to represent scale, rotation, and translation in an  $N$ -dimensional space, so we will be working with  $3 \times 3$  matrices in this section.

We turn a transformation formula into a transformation matrix by placing the multiplicative factors and translational component for the  $x$  coordinate



in the first row, and the factors and translation for the y coordinate in the second row. Any element along the diagonal that doesn't already have a value is set to 1, while any such element off the diagonal is set to 0.

We can write scaling as a matrix like this:

$$S = \begin{bmatrix} \vec{s}_x & 0 & 0 \\ 0 & \vec{s}_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

where  $s_x$  is the scale factor along the x axis, and  $s_y$  is the scale factor along the y axis.

We apply a transformation to a point by treating the vector as a *column matrix* with dimensions  $3 \times 1$  with a 1 in the bottom-most row. For example, to scale a vector with our newly-constructed 2D scaling matrix, we'd write this:

$$v = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$Sv = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} v_x \\ v_y \\ 1 \end{bmatrix} = \begin{bmatrix} s_x v_x \\ s_y v_y \\ 1 \end{bmatrix}$$

Note that the x coordinate is scaled by the x scaling factor, and the y coordinate is scaled by the y scaling factor, as intended.

We can formulate a rotation matrix using the same procedure as above, using the sin and cosine factors we learned about earlier:

$$R = \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

By multiplying this matrix with a vector to the right, we recover our earlier rotation equations:

$$Rv = \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} v_x \\ v_y \\ 1 \end{bmatrix} = \begin{bmatrix} v_x \cos \theta - v_y \sin \theta \\ v_x \sin \theta + v_y \cos \theta \\ 1 \end{bmatrix}$$

Finally, we can bring translation into this framework by placing the translation offsets into the third column.

$$T = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix}$$

When we perform matrix multiplication with a translation matrix, the final

<sup>1</sup> in our point's column matrix “picks up” the translational components, resulting in them being added to the x and y coordinates respectively:

$$Tv = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} v_x \\ v_y \\ 1 \end{bmatrix} = \begin{bmatrix} v_x + t_x \\ v_y + t_y \\ 1 \end{bmatrix}$$

This is exactly what we expect from translation.

Now we can write a single matrix that scales, rotates, and translates a point all at once.

$$v' = TRSv$$

With that, we've successfully established a mathematical framework for treating all basic transformations as matrices, which allows us to easily compose arbitrary sequences of transformations together via matrix multiplication.

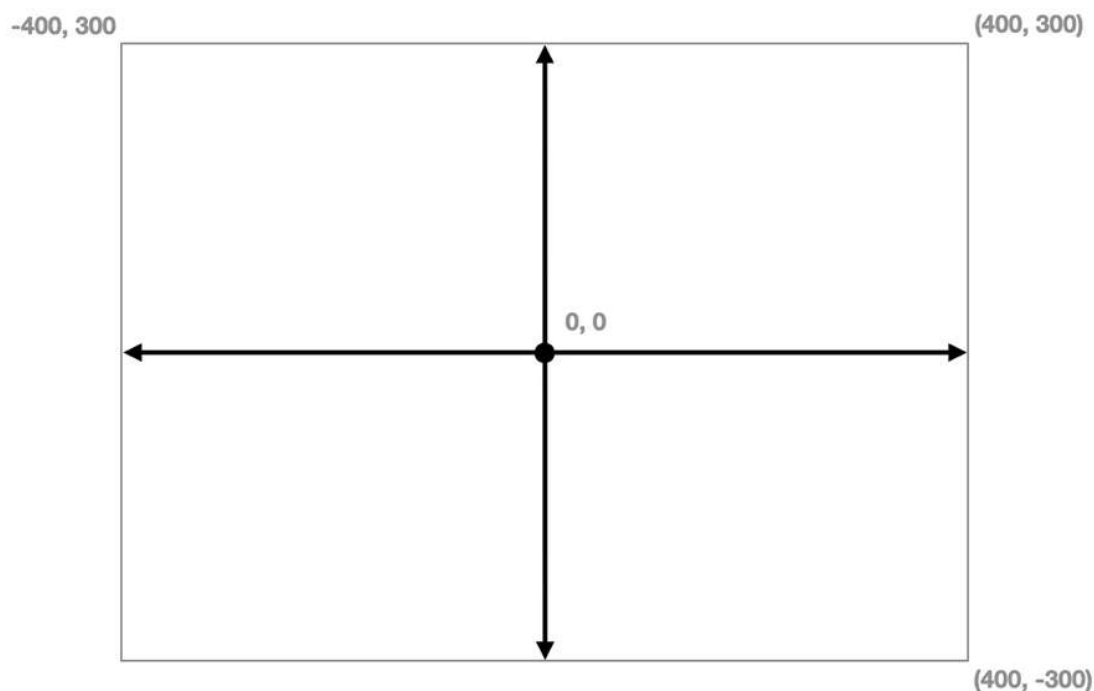
## ***Virtualizing the Canvas***

Recall again that we've been writing all of our vertex positions in terms of normalized device space. This has been convenient because it means we can pass these positions directly through our vertex function. However, it is often not the most convenient space to work in.

Ordinarily, in a painting program like Adobe Photoshop, you define a canvas size that lets you specify how large the image should be in pixels. The image might later be scaled and zoomed in a different context, but these operations happen relative to the dimensions you originally chose.

We can do the same thing when drawing pictures with Metal. Imagine we have a window that is 800×600 points in size. Then, instead of working in a space whose x coordinates run from -1 on the left to 1 on the right, we might choose a space that runs from -400 to 400, for a total width of 800,

matching our window.



Since the user might choose to resize the window, we need to decide how to react. The right answer depends on context, but one thing we could choose to do is scale the image to continue filling the view, while resizing it in the vertical dimension to avoid distorting shapes unnecessarily.

Whatever we decide to do, we need a way to transform from our chosen canvas space to NDC space.

## ***Projection Transformations***

The task of reshaping our virtual world to match NDC is called *projection*. Projection simultaneously transforms vertices into NDC and introduces perspective, if necessary. We won't discuss perspective here just yet, but we do need to get comfortable with the idea of projection transformations.

The coordinates in the canvas space we chose above run from -400 to 400 in x and -300 to 300 in y by default. We can generalize this into a scheme where we specify arbitrary left, right, top, and bottom boundaries for our virtual canvas. Then, the task of formulating a projection transform amounts to scaling and shifting points on our canvas into normalized device coordinates.

## ***Normalized Device Coordinates, in Depth***

I haven't been entirely forthcoming about the nature of NDC space. In addition to x and y axes that run from -1 to 1, NDC space also has a z axis that runs from 0 to 1, pointing away from us, the viewer. To formulate a projection transform that does the right thing, we need to account for this third dimension.

Fortunately, because we're still drawing in 2D, we can mostly get away with ignoring the z axis. The z coordinate of all of our vertices will be forced to 0 by our vertex function, as we've been doing all along. So, all we have to do is pick a sensible range for z coordinates in our canvas space, and map it to NDC's z axis with our projection transformation.

As you may have guessed by now, we're going to build a matrix to do this transformation. I know I *just* introduced another dimension to our idea of NDC, but that implies that we also need to add another row and column to our transformation matrices, making them 4×4.

This isn't as bad as it sounds: our rotation, scale, and translation matrices just get a few more ones and zeroes, following all of the same rules as before:

$$S = \begin{bmatrix} \vec{s}_x & 0 & 0 & 0 \\ 0 & \vec{s}_y & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R = \begin{bmatrix} \cos \theta & -\sin \theta & 0 & 0 \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T = \begin{bmatrix} 1 & 0 & 0 & \vec{t}_x \\ 0 & 1 & 0 & \vec{t}_y \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

# Orthographic Projections

With our existing transformation matrices patched up for the third dimension, let’s continue exploring projection.

We’ll be doing a particular kind of projection called “orthographic projection.” The term orthographic here means that our virtual scene is projected along parallel lines toward the viewing plane. This contrasts with perspective projection, in which projection happens along non-parallel lines, introducing effects like foreshortening. We will look at perspective projection when we start drawing in 3D.

An orthographic projection transforms each axis using the same procedure: scale the coordinates so that they span the width of NDC, then offset them so they’re centered on the origin. If that sounds like a combination of a scale matrix and a translation matrix, that’s because it is.

We write our orthographic projection matrix in terms of the left, right, top, and bottom boundaries of our canvas. For brevity, they’re abbreviated as `l`, `r`, `t`, and `b` below. We also have near and far boundaries in z, abbreviated `n` and `f`, which we’ll just set to 0 and 1 for now.

$$P_{ortho} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & \frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & \frac{t+b}{b-t} \\ 0 & 0 & \frac{1}{n-f} & \frac{n}{n-f} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

## Matrix Math in Swift

Swift’s simd framework includes small matrix types that are ideal for expressing transformation matrices. In particular, the `simd_float4x4` type is a 4x4 matrix that includes operators for performing essential operations

like matrix-matrix multiplication and matrix-vector multiplication.

We will add extensions to the `simd_float4x4` matrix type for each of our basic transformations. Note that the initializer we delegate to expects to receive column vectors rather than row vectors, so if things seem flipped, that's why.

Here are our basic scale, rotate, and translate transformations in code:

```
self.init(SIMD4<Float>(s.x,    0  0)
           0,    ,    ,
           SIMD4<Float>( 0, s.y,  0  0)
           ,    ,
           SIMD4<Float>( 0,    0,  1  0)
           ,    ,
           SIMD4<Float>( 0,    0,  0  1)
           ,    )
```

```
extension simd_float4x4
  self.init(SIMD4<Float>( c, s,  0  0)
            ,    ,
            SIMD4<Float>(-s, c,  0  0)
            ,    ,
            SIMD4<Float>( 0, 0,  1  0)
            ,    ,
            SIMD4<Float>( 0, 0,  0  1)
            ,    )

  init(translate2D t: {
    SIMD2<Float>)
  self.init(SIMD4<Float>( 1,    0  0,  0)
            ,    ,
            SIMD4<Float>( 0,    1  0,  0)
            ,    ,
            SIMD4<Float>( 0,    0  1,  0)
            ,    ,
            SIMD4<Float>(t.x, t.y, 0, 1))
}
```

We can also write an orthographic projection matrix extension using the formula from above.

```

extension simd_float4x4 {
    init(orthographicProjectionWithLeft left: Float, top: Float,
        right: Float, bottom: Float, near: Float, far: Float)
    {
        let sx = 2 / (right - left)
        let sy = 2 / (top - bottom)
        let sz = 1 / (near - far)
        let tx = (left + right) / (left - right)
        let ty = (top + bottom) / (bottom - top)
        let tz = near / (near - far)
        self.init(SIMD4<Float>(sx, 0, 0, 0),
            SIMD4<Float>(0, sy, 0, 0),
            SIMD4<Float>(0, 0, sz, 0),

            SIMD4<Float>(tx, ty, tz, 1))
    }
}

```

## ***Matrix-Vector Math in Shaders***

Since we can combine multiple transformations (including projection) into a single matrix, adapting our vertex function to use transformations is very straightforward.

As before, we take a constant buffer parameter, but this time it is of type

`float4x4 &`, a reference to a single 4x4 matrix.

We then rewrite our position transformation expression to use the more general transformation matrix:

```

out.position = transform * float4(in.position, 0.0, 1.0);

```

The transform matrix here could have any number of transformations baked into it, and we'll explore some time-based animations below using this flexibility.

## ***Animating Transformations in Time***

To conclude this article, let's put everything into practice by modifying our `updateConstants()` method to generate a transformation matrix that combines animating scaling, rotation, and translation effects with our orthographic projection.

We start by defining a new member variable in our renderer to keep track of elapsed time.



```
var time: TimeInterval = 0.0
```

We now rewrite our `updateConstants()` method in its entirety, starting with

some timekeeping:

```
func updateConstants() {  
    time += 1.0 / Double(view.preferredFramesPerSecond)  
    let t = Float(time)
```

The first effect we'll introduce is a pulsating scale effect, which will grow and shrink the triangle around its center:

```
let pulseRate: Float = 1.5  
let scaleFactor = 1.0 + 0.5 * cos(pulseRate * t)  
let scale = SIMD2<Float>(scaleFactor, scaleFactor)  
let scaleMatrix = simd_float4x4(scale2D: scale)
```

Next, we apply a rotation that spins the triangle about its center:

```
let rotationRate: Float = 2.5  
let rotationAngle = rotationRate * t  
let rotationMatrix = simd_float4x4(rotateZ: rotationAngle)
```

Finally, we will apply a translation effect that moves the triangle around the screen in a circle, much as we did last time when introducing constant buffers:

```
let orbitalRadius: Float = 200  
let translation = orbitalRadius * SIMD2<Float>(cos(t), sin(t))  
let translationMatrix = simd_float4x4(translate2D: translation)
```

We combine these three matrices into a so-called model matrix, which transforms the triangle from its own local coordinate space into our virtual canvas space.

```
let modelMatrix = translationMatrix * rotationMatrix * scaleMatrix
```



To build our orthographic projection matrix, we calculate a canvas size that is 800 units wide and whose height adapts to the aspect ratio of the view.

The center of the view coincides with the origin of our canvas space.

```
let aspectRatio = Float(view.drawableSize.width /
view.drawableSize.height)
    let canvasWidth: Float = 800
    let canvasHeight = canvasWidth / aspectRatio
    let projectionMatrix =
simd_float4x4(orthographicProjectionWithLeft: -canvasWidth / 2,
                                                    top: canvasHeight / 2,
                                                    right: canvasWidth / 2,
                                                    bottom: -canvasHeight / 2,
                                                    near: 0.0,
                                                    far: 1.0)
```

To get the final transformation matrix, we multiply our projection matrix by the model matrix of the triangle:

```
var transformMatrix = projectionMatrix * modelMatrix
```

Now, using the dynamic constants technique from the previous article, we can send our transformation matrix to the GPU to be applied in the vertex function:

```
currentConstantBufferOffset =
(frameIndex % MaxOutstandingFrameCount) * constantsStride

let constants = constantBuffer.contents()
    .advanced(by: currentConstantBufferOffset)

constants.copyMemory(from: &transformMatrix,
                    byteCount: constantsSize)
}
```

(Note that we have updated the constants size member to match the size of the transform matrix, `MemoryLayout<simd_float4x4>.size`.)

With these changes in place, we can run the app and watch our triangle pulse, rotate, and tumble around the screen.





# Day 11: Meshes



Warren Moore ·

9 min read · Apr 12, 2022

*This series of posts is my attempt to present the Metal graphics programming framework in small, bite-sized chunks for Swift app developers who haven't done GPU programming before.*

*If you want to work through this series in order, start [here](#).*

If you're anything like me, you're probably tired of drawing a single triangle. Our triangle friend has served us well through our exploration of draw calls, vertex attributes, dynamic constants, and some pretty heady math.

It's time to move on. This article will introduce the concept of the mesh, a unit of geometry that can be drawn with a single draw call. In addition to seeing how to draw a lot more triangles, we will see how to use indexing to store such geometry much more efficiently.

## ***Primitives, Connectivity, and Meshes***

Unlike most graphics libraries, Metal doesn't provide high-level APIs for drawing curves, text, or polygons. Everything we draw with Metal has to be broken down into constituent parts called *primitives*.

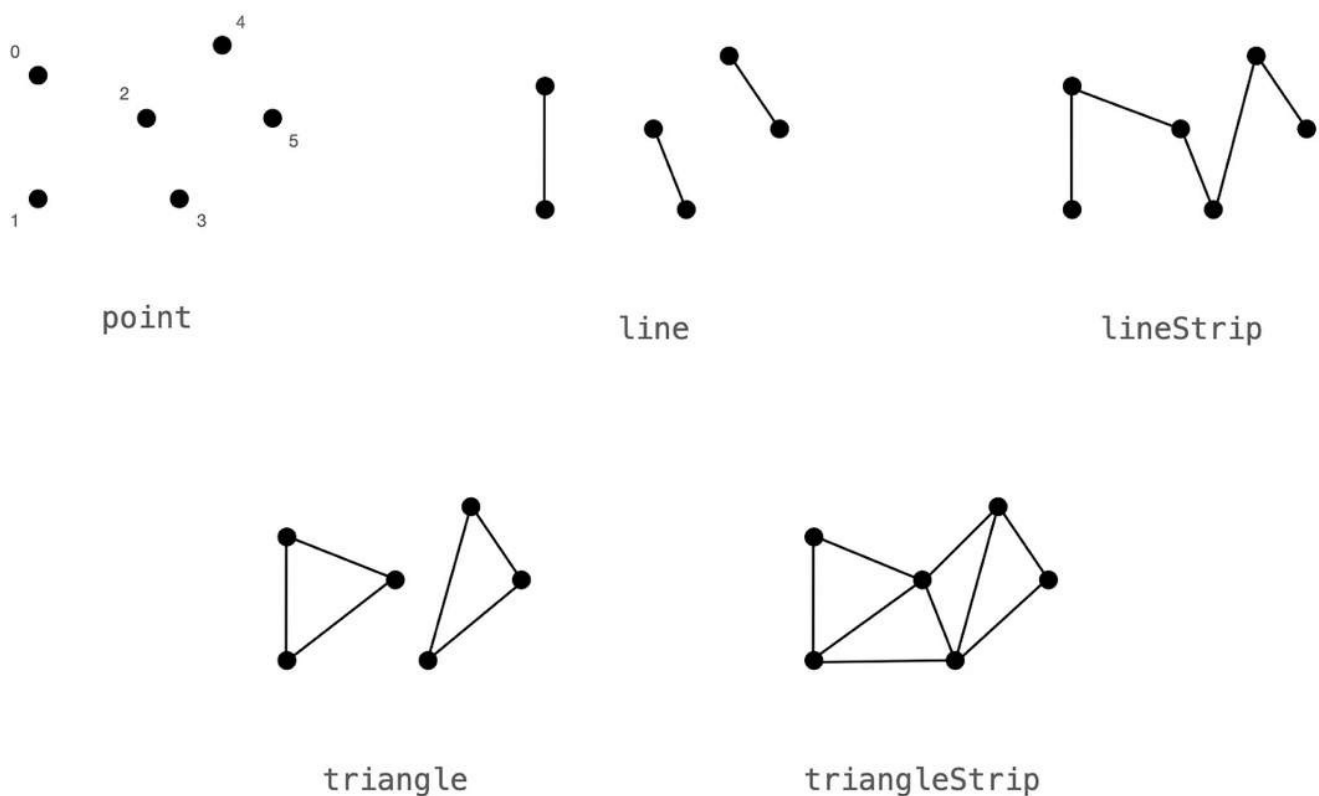
We have already become familiar with one Metal primitive: the triangle. However, there are other primitive types we should get acquainted with.

Looking at the declaration of `MTLPrimitiveType` indicates there are five basic primitives we can draw with Metal:

```
enum MTLPrimitiveType : UInt
{ case point = 0
  case line = 1
  case lineStrip = 2
  case triangle = 3
  case triangleStrip = 4
}
```

The primitive type of a draw call indicates how vertices are combined together into these shapes. In the case of `.point`, each vertex is rendered as a small screen-aligned square. The `.line` primitive takes pairs of vertices and joins them into line segments. A line strip (`.lineStrip`) is a connected sequences of line segments, where each vertex after the first one defines the endpoint of the next segment. A triangle (`.triangle`) is a polygon that consists of three vertices. A triangle strip (`.triangleStrip`) is a connected group of triangles, where each vertex after the first two defines a triangle along with the previous two vertices.

These primitive types are shown in the figure below, which also indicates how the numbered vertices would be connected together to form each type of primitive.



We call the way that vertices are joined together to make primitives their



*connectivity*. By default, the order of vertices in the vertex buffer is used to

determine which vertices are joined together, but we will see a technique below for influencing this.

A *mesh*, (or *polygon mesh*) is the set of vertices, edges, and faces that comprise a 2D or 3D object. Let's take a look at how we represent this abstract notion in code.

## ***A Simple Mesh Class***

Since we want to expand the abilities of our renderer beyond a single triangle, it is natural to create an abstraction that will help us manage many triangle as a unit.

We will call our basic mesh class `SimpleMesh`. The main purpose of this class is to gather up all of the data we use in a draw call into a single object.

For the time being, we will make the assumption that a mesh consists only of triangle data. A mesh can have multiple vertex buffers, each holding the data for one or more attributes. The layout of these buffers is described by the vertex descriptor. We also store the mesh's vertex count.

```
class SimpleMesh {
  let vertexBuffers: [MTLBuffer]
  let vertexDescriptor: MTLVertexDescriptor
  let vertexCount: Int
  let primitiveType: MTLPrimitiveType = .triangle
  // ...
}
```

We define an initializer that sets up the members of the mesh:

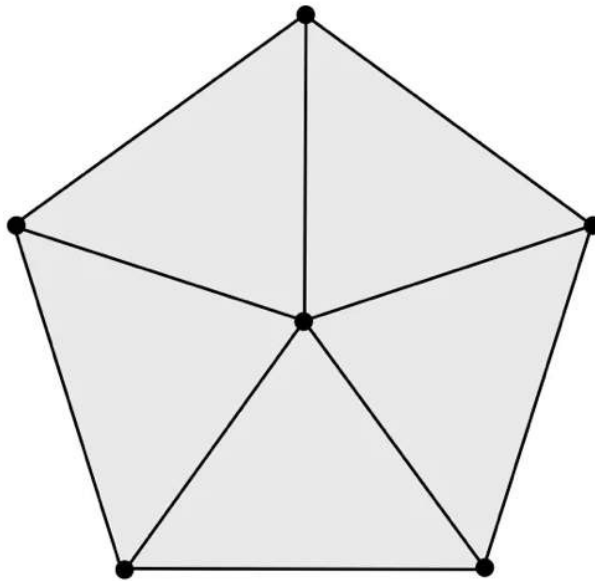
```
init(vertexBuffers: [MTLBuffer],
     vertexDescriptor: MTLVertexDescriptor,
     vertexCount: Int)
```

This mesh class is indeed quite simple so far, but it has everything we need

to start defining polygon meshes that we can draw with Metal.

## Generating Meshes Programmatically

Suppose we want to provide a convenience initializer that creates a regular polygon mesh, such as this pentagon:



To draw polygons with Metal, we need to decompose them into triangles. In this case, the pentagon consists of five triangles. Since the vertices (except for the center) are evenly spaced along the circumference of a circle, we can generate their positions in a loop using trigonometric functions.

For a regular polygon with a given radius and side count, we can enumerate the vertex positions like this:

```
var angle: Float = .pi / 2
let deltaAngle = (2 * .pi) / Float(sideCount)
for _ in 0..
```

Our initializer will take the polygon side count, radius, a solid color, and a device (which will allow us to allocate vertex buffers).

```
convenience init(planarPolygonSideCount sideCount: Int,
```



```
radius: Float,  
color: SIMD4<Float>,  
device: MTLDevice)
```

Unlike in previous samples, we will store the positions and colors in separate buffers, so we declare two arrays to accumulate them:

```
{  
    var positions = [SIMD2<Float>]()  
    var colors = [SIMD4<Float>]()  
    // ...
```

We then generate our positions and colors using a loop like the one sketched out previously. Since each side of the polygon corresponds to a triangle, we generate three vertices per loop iteration:

```
var angle: Float = .pi / 2  
let deltaAngle = (2 * .pi) / Float(sideCount)  
for _ in 0..  
    positions.append(SIMD2<Float>(radius * cos(angle),  
                                   radius * sin(angle)))  
    colors.append(color)  
  
    positions.append(SIMD2<Float>(radius * cos(angle + deltaAngle),  
                                   radius * sin(angle + deltaAngle)))  
    colors.append(color)  
  
    positions.append(SIMD2<Float>(0, 0))  
    colors.append(color)  
  
    angle += deltaAngle  
}
```

Now that we have our positions and colors, we can allocate some vertex buffers to store them, following a now-familiar pattern:

```
let positionBuffer =  
    device.makeBuffer( bytes:  
        positions,  
        length: MemoryLayout<SIMD2<Float>>.stride * positions.count,  
        options: .storageModeShared) !  
  
let colorBuffer =  
    device.makeBuffer( bytes: colors,  
        length: MemoryLayout<SIMD4<Float>>.stride * colors.count,  
        options: .storageModeShared) !
```



We finish up our convenience initializer by delegating to the initializer we previously declared:

```
self.init(vertexBuffers: [positionBuffer, colorBuffer],
          vertexDescriptor: SimpleMesh.defaultVertexDescriptor,
          vertexCount: positions.count)
}
```

The “default vertex descriptor” referenced above is a vertex descriptor that is common to all of the simple meshes we will build. It contains attributes for positions and colors, and two layouts that correspond to the two vertex buffers we allocate for each mesh:

```
private static var defaultVertexDescriptor: MTLVertexDescriptor
{ let vertexDescriptor = MTLVertexDescriptor()
  vertexDescriptor.attributes[0].format = .float2
  vertexDescriptor.attributes[0].offset = 0
  vertexDescriptor.attributes[0].bufferIndex = 0
  vertexDescriptor.attributes[1].format = .float4
  vertexDescriptor.attributes[1].offset = 0
  vertexDescriptor.attributes[1].bufferIndex = 1
  vertexDescriptor.layouts[0].stride =
MemoryLayout<SIMD2<Float>>.stride
  vertexDescriptor.layouts[1].stride =
MemoryLayout<SIMD4<Float>>.stride
  return vertexDescriptor
}
```

This is the first time we have used more than one vertex buffer, so it’s worth considering how this vertex descriptor differs from earlier ones.

## ***Updating the Renderer to Draw Meshes***

Now that our renderer no longer holds vertex buffers, we need to adapt its draw method to use meshes. Suppose our renderer has a member that holds a mesh:

```
let mesh: SimpleMesh
```



We can initialize a pentagon mesh when creating the renderer:

```
mesh = SimpleMesh(planarPolygonSideCount: 5,
                  radius: 250,
                  color: SIMD4<Float>(0.0, 0.5, 0.8, 1.0),
                  device: device)
```

When it comes time to draw, we need to bind all of the mesh's vertex buffers into different slots. We use the `enumerated()` method of `Array` to get an `(Int, MTLBuffer)` tuple for each buffer, then set it on the render command encoder:

```
for (i, vertexBuffer) in mesh.vertexBuffers.enumerated()
{ renderCommandEncoder.setVertexBuffer(vertexBuffer,
                                       offset: 0,
                                       index: i)
}
```

Since we are using multiple buffers, we need to shift the index of our constant buffer so it doesn't collide with our vertex buffers. We'll use buffer index 2 for now:

```
renderCommandEncoder.setVertexBuffer( constantBuffer,
                                       offset: currentConstantBufferOffset,
                                       index: 2)
```

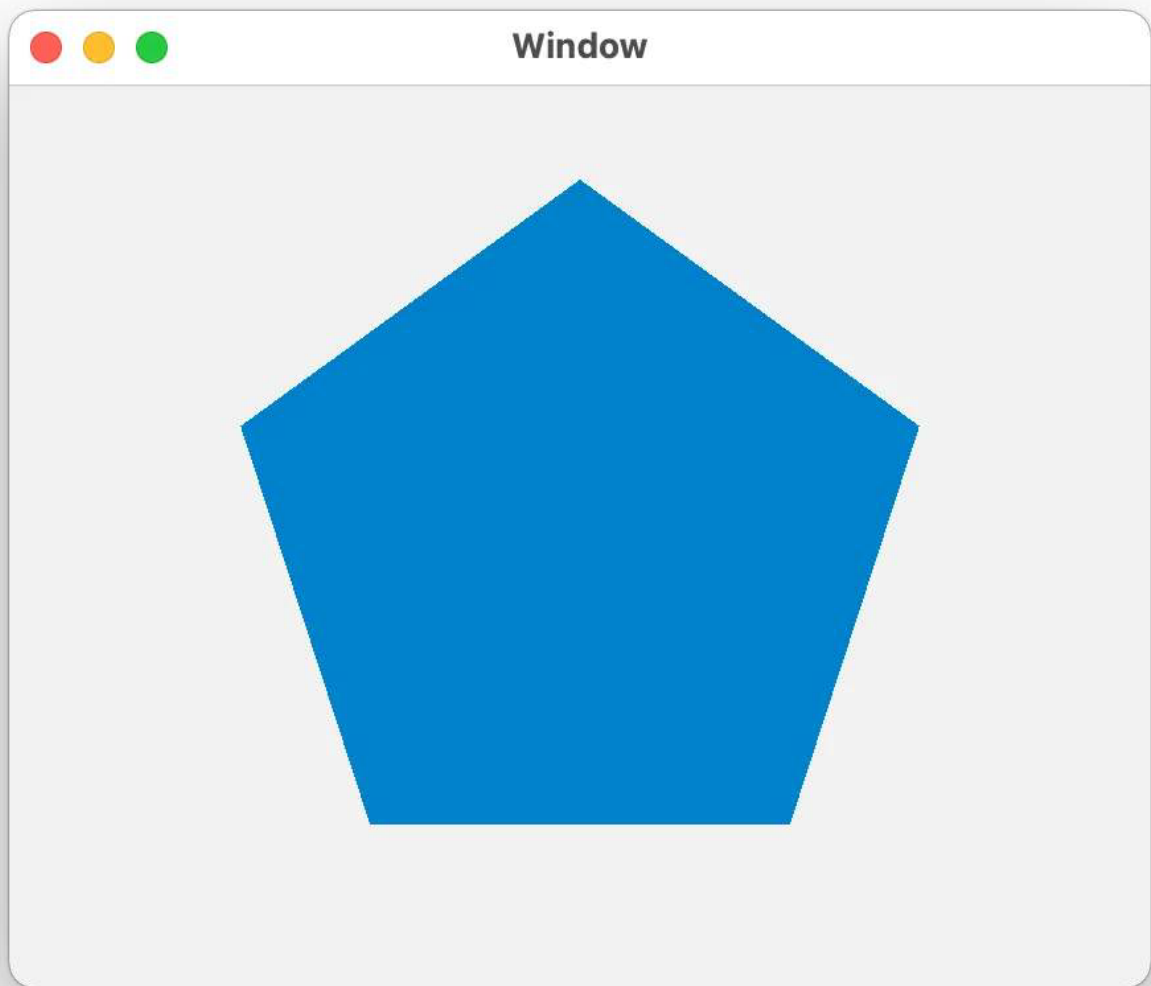
A corresponding change must be made in the shader source as well.

Finally, when drawing, we use the primitive type and vertex count provided by our mesh object:

```
renderCommandEncoder.drawPrimitives(type: mesh.primitiveType,
                                    vertexStart: 0,
                                    vertexCount: mesh.vertexCount)
```

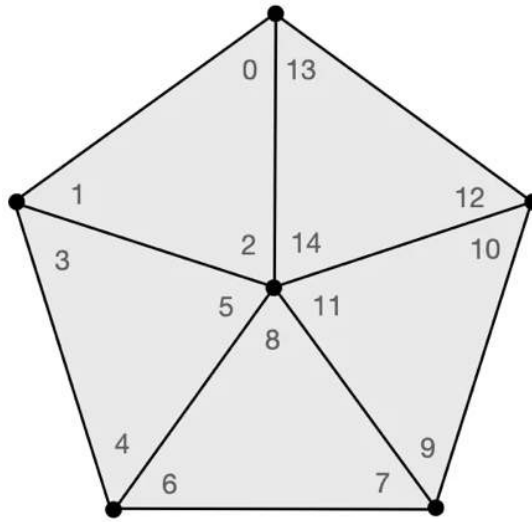
We have successfully created a simple class that allows us to store all of the data for a mesh together, which will be important as we scale up the

number and complexity of the objects we draw.



There is one minor problem that you might have noticed, however. Our polygon mesh has 15 vertices, but many of these vertices are exact duplicates. For example, the data for the center vertex is repeated five times in the buffers, which is somewhat wasteful.

This redundancy is illustrated below. Each vertex is labelled by its corresponding vertex ID. As you can see, each vertex is duplicated at least once.

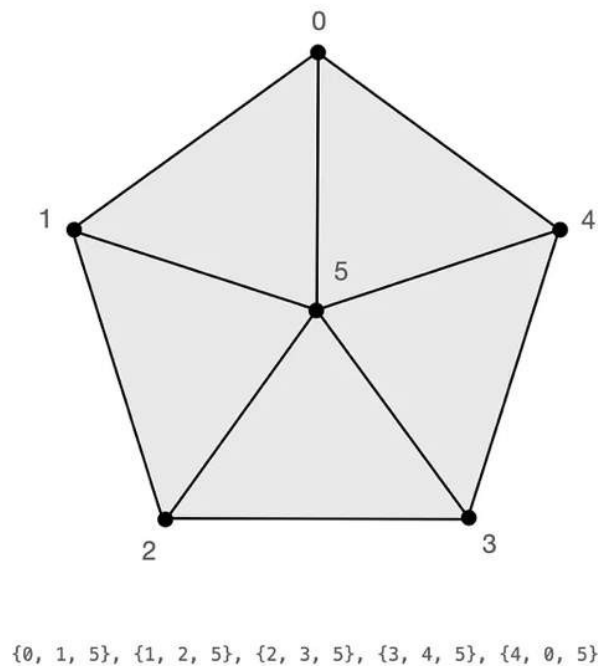


## ***Indexed Geometry***

Redundant data storage becomes a more severe problem as meshes get more complex, so we need a strategy for tackling it. We will use a feature of Metal called *indexed drawing*.

Instead of duplicating vertices each time we want to include them in a primitive, we store each vertex exactly once, and use an *index buffer* to indicate which vertices belong to which primitives.

Below is an illustration of how we organize data for indexed drawing. Each vertex now has a unique label, and the indices of the vertices of each triangle are listed below. Because indices tend to be much smaller than vertices, this scheme can save a lot of space.



Let's consider how we can extend our mesh class to support indexed drawing.

## ***Indexed Geometry in Metal***

Like vertices, indices are stored in buffers. We allocate index buffers using exactly the same `MTLDevice` methods we use to allocate vertex buffers. However, index buffers store different types of data.

Metal supports two index types: `uint16` and `uint32`. These two unsigned integer types are 2 bytes and 4 bytes in size, respectively. Before we create an index buffer, we need to know which of these types to use. `uint16` can hold up to 65,534 different values, so it is suited to small- to medium-sized meshes. `uint32` has an upper bound of several billion, so it will cover every other case.

We add the following members to the `SimpleMesh` class to allow it to optionally support indexed drawing:

```
let indexBuffer: MTLBuffer?
let indexType: MTLIndexType = .uint16
let indexCount: Int
```

Pretty straightforward. We store the index buffer and the type of indices it



holds, along with the number of indices in the buffer.

We also define another initializer that allows us to create an indexed mesh:

```
init(vertexBuffers: [MTLBuffer], vertexDescriptor:
MTLVertexDescriptor, vertexCount: Int,
    indexBuffer: MTLBuffer, indexCount: Int)
{
    self.vertexBuffers = vertexBuffers
    self.vertexDescriptor = vertexDescriptor
    self.vertexCount = vertexCount
    self.indexBuffer = indexBuffer
    self.indexCount = indexCount
}
```

## ***Generating Indexed Geometry***

Let's adapt our polygon mesh generator to make indexed geometry. The key idea is to only write each vertex into the vertex buffers once. Then, we can generate an index list that references the vertices efficiently.

The signature of the new convenience initializer looks similar to the previous one:

```
convenience init(
    indexedPlanarPolygonSideCount sideCount: Int,
    radius: Float,
    color: SIMD4<Float>,
    device: MTLDevice)
```

As before, we generate the vertices by looping around the polygon's enclosing circle. As a final step, we append the polygon's center vertex.

```
var positions = [SIMD2<Float>]()
var colors = [SIMD4<Float>]()

var angle: Float = .pi / 2
let deltaAngle = (2 * .pi) / Float(sideCount)
for _ in 0..
```





```

}
positions.append(SIMD2<Float>(0, 0))
colors.append(color)

```

The vertex buffers are then generated exactly as before.

To build the index list, we loop around the polygon’s perimeter, appending three indices per triangle. Since we’re not likely to generate a 20,000-gon, we’ll use the smaller index type, which is represented in Swift as `UInt16`.

```

var indices = [UInt16]()
let count = UInt16(sideCount)
for i in 0..

```

Creating an index buffer looks just like creating a vertex buffer:

```

let indexBuffer =
    device.makeBuffer( bytes: indices,
                      length: MemoryLayout<UInt16>.size * indices.count,
                      options: .storageModeShared)!

```

Now that we have all of the necessary data, we can build an indexed mesh by delegating to the indexed mesh initializer:

```

self.init(vertexBuffers: [positionBuffer, colorBuffer],
         vertexDescriptor: SimpleMesh.defaultVertexDescriptor,
         vertexCount: positions.count,
         indexBuffer: indexBuffer,
         indexCount: indices.count)

```

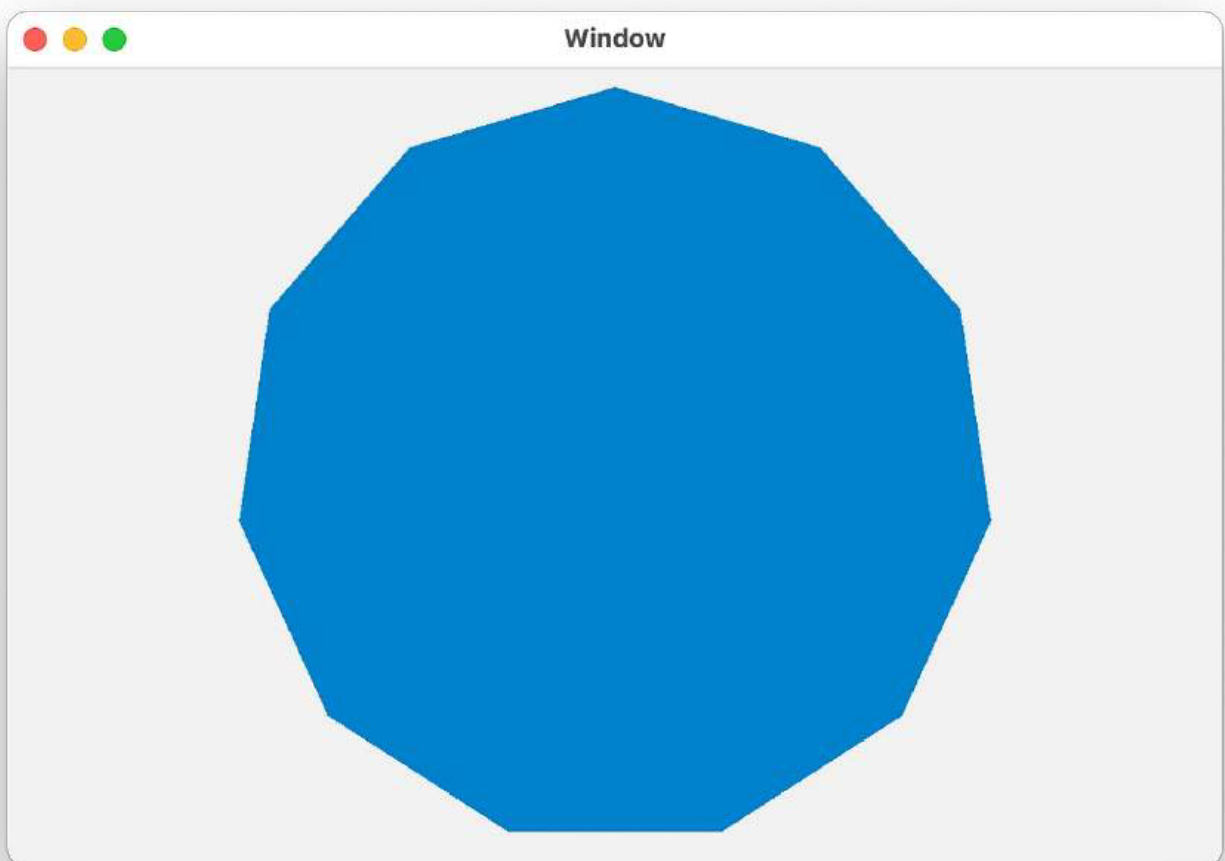
## ***Indexed Draw Calls***

Now we can create indexed meshes, but how do we draw them? Fortunately, the only difference between a non-indexed draw call and an indexed draw call is the method we call.

We call the `drawIndexedPrimitives(type:indexCount:indexType:indexBuffer:indexBufferOffset:)` method, providing the mesh values we used previously, along with the new ones we added for indexed drawing:

```
if let indexBuffer = mesh.indexBuffer
{ renderCommandEncoder.drawIndexedPrimitives(
    type: mesh.primitiveType,
    indexCount: mesh.indexCount,
    indexType: mesh.indexType,
    indexBuffer: indexBuffer,
    indexBufferOffset: 0)
}
```

Running the sample code with a side count of 11 produces the following picture:



Apparently an 11-sided polygon is called a hendecagon. Who knew?

Next time, we will revisit the MetalKit framework and take a look at some utilities it provides for generating meshes.

# Day 12: MDLMesh and MTKMesh



Warren Moore ·

8 min read · Apr 13, 2022

*This series of posts is my attempt to present the Metal graphics programming framework in small, bite-sized chunks for Swift app developers who haven't done GPU programming before.*

*If you want to work through this series in order, start [here](#).*

In the previous article, we looked at how to create a mesh class that holds the vertex buffers and index buffer needed to issue a draw call.

In this article, we will look at alternative abstractions provided by Model I/O and MetalKit, system frameworks that provide higher-level utilities that interoperate with Metal.

Why did we go through the rigamarole of defining our own abstraction when one already exists? In my experience, the mesh abstractions we will use in this article may not be the best way to organize mesh data in larger systems. We aim to create abstractions that serve our needs, so considering the design and tradeoffs of alternatives is always a useful exercise.

## **Introducing Model I/O and MDLMesh**

Model I/O is a framework introduced with iOS 9 to support importing and exporting 3D assets. It also includes utilities for generating various common types of meshes and textures.

The `MDLMesh` class in Model I/O is an abstraction of a mesh. Similar to our simple mesh, it contains an array of vertex buffers. An `MDLMesh` can also contain one or more *submeshes*, where each submesh has its own list of

indices (index buffer). This allows a mesh to contain multiple disjoint parts that reference the same vertex data.

To get more familiar with the interfaces we will use, here is a simplified class definition for `MDLMesh`:

```
class MDLMesh {
    var vertexDescriptor: MDLVertexDescriptor
    var vertexCount: Int
    var vertexBuffers: [MDLMeshBuffer]
    var submeshes: NSMutableArray?
    var allocator: MDLMeshBufferAllocator
}
```

Here is a simplified definition of `MDLSubmesh`:

```
class MDLSubmesh {
    var indexBuffer: MDLMeshBuffer
    var indexCount: Int
    var indexType: MDLIndexBitDepth
    var geometryType: MDLGeometryType
}
```

Considering `MDLMesh` and `MDLSubmesh` together, we can see that their member variables are very similar to those of our own mesh class.

## ***Model I/O Buffer Allocation***

One difference between our own mesh class and `MDLMesh` is the `allocator` member of `MDLMesh`. What is the purpose of the `MDLMeshBufferAllocator` protocol?

Model I/O doesn't draw anything or interact with the GPU itself: it is “graphics API-agnostic.” You might draw a Model I/O mesh with Metal, or you might save it to disk, or you might convert it into another format suitable for printing on a 3D printer.

In each of these use cases, model data might be stored differently. In particular, if we want to draw a mesh with Metal, we know we eventually

need its vertex data to be in `MTLBuffer`s.

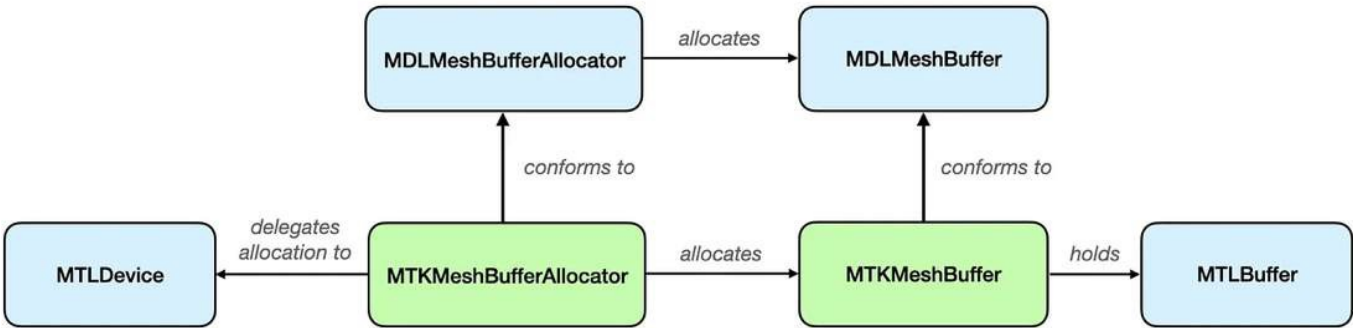
For this reason, Model I/O defers to the API user to provide a strategy object that conforms to the `MDLMeshBufferAllocator` protocol. The allocator produces mesh buffers, which are objects conforming to the `MDLMeshBuffer` protocol. Model I/O can write into these buffers without having to know whether they were allocated with `malloc`, with an `MTLDevice`, or in some other way.

## MetalKit Buffer Allocation

We want to avoid copying mesh data, since excessive copies waste memory and CPU cycles. To ensure that Model I/O can write directly into GPU-visible memory, the MetalKit framework provides its own Model I/O-compatible allocator type:

`MTKMeshBufferAllocator`.

All of these type relationships can be a little confusing, so here's a diagram showing how the various Model I/O, MetalKit, and Metal types interact:



The key idea is that `MTKMeshBufferAllocator` implements the `MDLMeshBufferAllocator` interface to abstract GPU buffer creation away from Model I/O.

We create a MetalKit buffer allocator by providing the device to which it will delegate buffer creation:

```
let allocator = MTKMeshBufferAllocator(device: device)
```



## Creating Meshes with Model I/O

Now that we have a buffer allocator, we can ask Model I/O to create meshes for us.

`MDLMesh` has a number of methods for creating various basic geometry types: spheres, planes, cylinders, etc.

We create a sphere mesh by specifying the extents of its bounding box (width, height, and depth) and the number of segments it should be divided into (longitude-wise and latitude-wise). We can also indicate that we want the faces of the sphere to point inwardly by passing `true` for the `inwardNormals` parameter. Finally, we supply the mesh buffer allocator we created previously.

```
let mdlMesh = MDLMesh(sphereWithExtent: SIMD3<Float>(1, 1, 1),
                      segments: SIMD2<UInt32>(24, 24),
                      inwardNormals: false,
                      geometryType: .triangles,
                      allocator: allocator)
```

The initial layout of the mesh data is determined by Model I/O, but we can update it using a Model I/O vertex descriptor.

```
let vertexDescriptor = MDLVertexDescriptor()
```

The `MDLVertexDescriptor` type is similar in purpose to the `MTLVertexDescriptor` type we've used previously, but adds *semantic* information to attributes: each attribute has a name that indicates whether it holds position data, color data, and so on.

We'll ask Model I/O to generate positions and normals and place them together into a single buffer. A *normal* is a vector that is perpendicular to the mesh's surface at a particular location. We will use these normals when implementing lighting later.

Populating a Model I/O vertex descriptor looks very similar to our prior usage of Metal vertex descriptors:



```
vertexDescriptor.vertexAttributes[0].name =
    MDLVertexAttributePosition
vertexDescriptor.vertexAttributes[0].format = .float3
vertexDescriptor.vertexAttributes[0].offset = 0
vertexDescriptor.vertexAttributes[0].bufferIndex = 0

vertexDescriptor.vertexAttributes[1].name =
    MDLVertexAttributeNormal
vertexDescriptor.vertexAttributes[1].format = .float3
vertexDescriptor.vertexAttributes[1].offset = 12
vertexDescriptor.vertexAttributes[1].bufferIndex = 0

vertexDescriptor.bufferLayouts[0].stride = 24
```

We can now ask Model I/O to lay out the sphere’s vertex buffers as we prefer by setting its `vertexDescriptor` property:

```
mdlMesh.vertexDescriptor = vertexDescriptor
```

## ***Bringing Model I/O Data into MetalKit***

Model I/O’s interfaces are fairly abstract; we can’t use `MDLMesh` and `MDLSubmesh` directly with Metal to draw a 3D mesh. Instead, MetalKit provides yet another bridge between Metal and Model I/O: the `MTKMesh` class.

Here is a simplified definition of the `MTKMesh` class and its corresponding submesh class, `MTKSubmesh`:

```
class MTKMesh {
    var vertexBuffers: [MTKMeshBuffer]
    var vertexDescriptor: MDLVertexDescriptor
    var submeshes: [MTKSubmesh]
    var vertexCount: Int
}

class MTKSubmesh {
    var primitiveType: MTLPrimitiveType
    var indexType: MTLIndexType
    var indexBuffer: MTKMeshBuffer
    var indexCount: Int
}
```

Ah, good. Yet another mesh abstraction that is similar to, yet meaningfully

distinct from the ones we've seen before. The useful thing to notice here is that we're finally seeing Metal types in this interface, instead of their more abstract Model I/O counterparts. This should give us hope that we'll actually be drawing something soon.

We can get an `MTKMesh` from an `MDLMesh` by using `MTKMesh(mesh:device:)` initializer:

```
mesh = try MTKMesh(mesh: mdlMesh, device: device)
```

This initializer can throw, so this code should be wrapped in a `do-catch` block, or we can choose to halt exception if an error occurs, by using the `try!` keyword.

We can modify our renderer class by replacing our earlier `SimpleMesh` member variable with an `MTKMesh` variable:

```
var mesh: MTKMesh!
```

Then we can add the mesh creation code from above to our `makeResources()` method.

## ***Adapting the Vertex Function***

We need to make some small updates to our vertex function to account for the fact that we're now storing vertex normals instead of colors. We update the vertex structures with a normal member, replacing the color member:

```
struct VertexIn {
    float3 position [[attribute(0)]];
    float3 normal    [[attribute(1)]];
};

struct VertexOut {
    float4 position [[position]];
    float3 normal;
};
```



Now that the incoming vertex position has 3 components, we stop forcing its z component to 0. We continue setting the position's fourth component to 1 so that our transformation matrices behave correctly:

```
vertex VertexOut vertex_main(
    VertexIn in [[stage_in]],
    constant float4x4 &transform [[buffer(2)]]
)
{
    VertexOut out;
    out.position = transform * float4(in.position, 1.0);
    out.normal = in.normal;
    return out;
}
```

## Adapting the Fragment Function

We will look much more closely at lighting in future articles. Let's get a glimpse of what's to come up implementing the simplest possible lighting shader.

The `normal` member of our interpolated fragment data tells us which way the surface faces at each point. We can use this to cheaply compute roughly how much the surface is oriented toward a virtual light source. It turns out that the amount of light reflected by a surface is proportional to the *dot product* between the light direction and the surface direction (normal).

Again, we won't go deeper into the theory for now, but here is a fragment function that implements this rudimentary lighting model:

```
fragment float4 fragment_main(VertexOut in [[stage_in]])
{
    float3 L = normalize(float3(1, 1, 1));
    float3 N = normalize(in.normal);
    float NdotL = saturate(dot(N, L));
    return float4(float3(NdotL), 1);
}
```

## Adapting the Render Pipeline

We need to update the vertex descriptor we supply when creating our

render pipeline. The easiest way to do this is with the

`MTKMetalVertexDescriptorFromModelIO` utility function, which converts an `MDLVertexDescriptor` to an equivalent `MTLVertexDescriptor`:

```
let vertexDescriptor =  
    MTKMetalVertexDescriptorFromModelIO(mesh.vertexDescriptor)!  
  
renderPipelineDescriptor.vertexDescriptor = vertexDescriptor
```

## ***Drawing an MTKMesh***

To draw a MetalKit mesh, we start as we did with our own mesh class: bind the vertex buffers to their respective locations.

```
for (i, meshBuffer) in mesh.vertexBuffers.enumerated()  
{ renderCommandEncoder.setVertexBuffer(  
    meshBuffer.buffer,  
    offset: meshBuffer.offset,  
    index: i)  
}
```

Note that the `MTKMeshBuffer` type has an `offset` member that indicates where in the buffer the vertex data begins. We need to use this when binding the buffer to get correct results.

We now iterate over the submeshes of the mesh, encoding one indexed draw command for each:

```
for submesh in mesh.submeshes {  
    let indexBuffer = submesh.indexBuffer  
    renderCommandEncoder.drawIndexedPrimitives(  
        type: submesh.primitiveType,  
        indexCount: submesh.indexCount,  
        indexType: submesh.indexType,  
        indexBuffer: indexBuffer.buffer,  
        indexBufferOffset: indexBuffer.offset)  
}
```

A submesh's index buffer is also an instance of `MTKMeshBuffer`, and so we also

supply its offset to the draw call, instead of passing 0 as we did



previously.

## ***Revising the Projection Matrix***

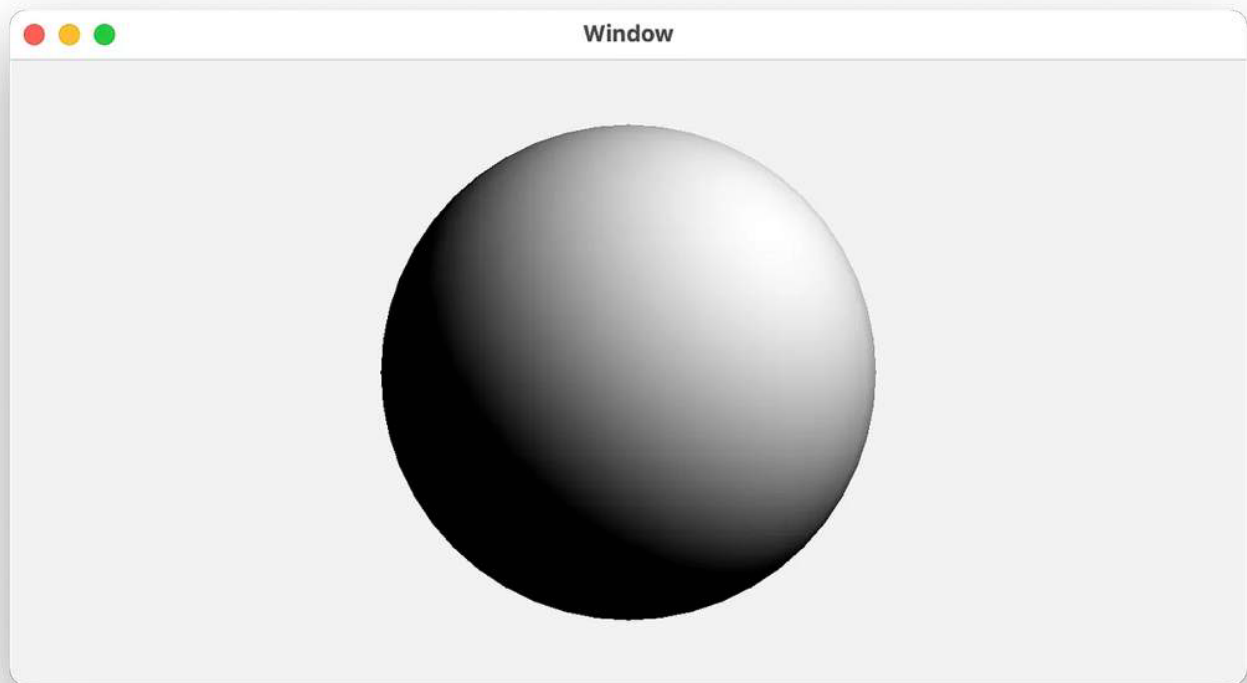
Since our sphere mesh is only one unit wide, we should choose a suitable projection transform to make it visible on the screen. I selected a constant scene width of 5 units and also expanded the z range to ensure the sphere's vertices were completely within the viewable volume. Those changes are shown below.

```
let aspectRatio = Float(view.drawableSize.width /
view.drawableSize.height)
let canvasWidth: Float = 5.0
let canvasHeight = canvasWidth / aspectRatio
let projectionMatrix =
simd_float4x4(orthographicProjectionWithLeft: -canvasWidth / 2,
               top: canvasHeight / 2,
               right: canvasWidth / 2,
               bottom: -canvasHeight / 2,
               near: -1,
               far: 1)
```

The code to update the dynamic constant buffer is the same as before.

## ***Conclusion***

If we run our updated sample app, we see something tantalizingly close to a 3D figure:



Despite the illusion afforded by our simple lighting shader, we would quickly experience issues if we tried to move around in this scene. Next time, we will finally take our first real step into 3D by discussing the depth buffer.

## ***Addendum***

The Model I/O framework can be awkward to use from Swift, because some of its types were imperfectly annotated. In particular, this means that some of the members of the `MDLVertexDescriptor` class have the wrong type. I prefer to fix this by adding the following extensions to the class:

```
extension MDLVertexDescriptor {
    var vertexAttributes: [MDLVertexAttribute]
    { return attributes as!
      [MDLVertexAttribute]
    }

    var bufferLayouts: [MDLVertexBufferLayout]
    { return layouts as!
      [MDLVertexBufferLayout]
    }
}
```

# Day 13: Depth



Warren Moore ·

5 min read · Apr 14, 2022

*This series of posts is my attempt to present the Metal graphics programming framework in small, bite-sized chunks for Swift app developers who haven't done GPU programming before.*

*If you want to work through this series in order, start [here](#).*

In the previous article, we looked at how to generate and draw 3D meshes with Model I/O and MetalKit. At long last, we're ready to start drawing 3D objects in earnest.

## **Thinking Like a Painter**

Up to this point, we haven't thought much about what happens along the z axis. When we start drawing overlapping geometry, though, we have to consider how things are ordered along the line of sight.

Consider how an oil painter builds up a scene, first painting the visible portions of the background, then painting objects nearer the point of view. We could take a similar approach when drawing our virtual 3D scenes: make a list of the triangles in our scene, sorting them so they are ordered from far to near, then draw them in order, replacing farther pixels with nearer pixels.

This approach, called the *painter's algorithm*, works as long as there are no ambiguities caused by intersecting or overlapping triangles. But there are two problems. First, there often isn't a way to unambiguously sort the triangles in a scene. Second, we might need to re-sort the triangle list every time an object or the point of view moves, and this is expensive.

## Depth Buffering

The problems inherent in the painter’s algorithm are resolved by considering the question of depth at each pixel. We use a separate texture called the *depth texture* (commonly called the *depth buffer*) to store the depth of the closest fragment we’ve seen so far when drawing. When a fragment is rasterized, we check its corresponding pixel in the depth buffer and replace the values in both the depth texture and color texture if the current fragment is closer to the point of view.

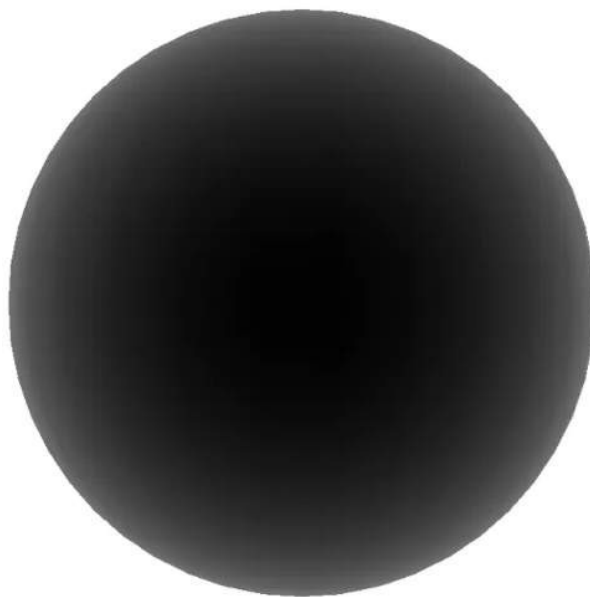
### The Depth Buffer in Metal

Metal has several possible pixel formats for depth textures. We will look at pixel formats in more detail when we discuss texturing. For the moment, we can ask our `MTKView` to create a depth texture for us by setting its `depthStencilPixelFormat` property to `MTLPixelFormat.depth32Float`.

```
view.depthStencilPixelFormat = .depth32Float
```

Each pixel of the resulting depth texture will store a 32-bit floating-point value between 0 and 1 representing the relative distance to the nearest object. The depth texture will be cleared to 1.0 each frame by default, so this value represents the absence of objects (the nearest object is “infinitely far away”).

As we draw objects, they replace the values in the depth buffer. Below is a visualization of a depth texture after a sphere has been drawn. Note the contrast between the center and edges of the sphere: darker values are closer.



## ***Depth-Stencil States***

Configuring an `MTKView` to produce a depth texture causes the view to populate the depth attachment of the render pass descriptors it vends. This tells Metal that a depth texture is available but is not sufficient for it to be used during rendering. We need to tell Metal to use the depth texture explicitly.

To do this, we create a *depth-stencil state* object. The depth-stencil state contains settings for configuring the GPU with our desired way of handling the depth buffer. There are two relevant bits of state we will consider here: enabling *depth-write*, and the *depth comparison function*.

To create a depth-stencil state, we first configure a *depth-stencil descriptor*:

```
let depthStencilDescriptor = MTLDepthStencilDescriptor()
```

We enable depth writing by setting `isDepthWriteEnabled` to true:



```
depthStencilDescriptor.isDepthWriteEnabled = true
```

We then configure the comparison function that should be used to determine whether a given fragment should overwrite the value already in the depth buffer. We already know that the depth buffer is cleared to 1.0, and than smaller values are closer, so we use `MTLCompareFunction.less`:

```
depthStencilDescriptor.depthCompareFunction = .less
```

Now that we have built our depth-stencil descriptor, we can hand it to a device and get back a depth-stencil state:

```
depthStencilState = device.makeDepthStencilState(descriptor:
depthStencilDescriptor)!
```

When rendering, we use our depth-stencil state by setting it on the render command encoder prior to issuing any draw calls:

```
renderCommandEncoder.setDepthStencilState(depthStencilState)
```

## ***Winding, Facing, and Culling***

There are two other bits of render state that become relevant when we start drawing in 3D: front-face winding and cull mode.

Given three vertices connected into a triangle, it is ambiguous which side is the “front” without more context. Rotating the triangle 180 degrees around the y axis shows a different side, but are we looking at the front or back now? We resolve this by saying that the front face of a triangle is

determined by the order in which its vertices are specified.

The choice or ordering is called the *winding order*, either *clockwise* or



*counterclockwise*. If we use clockwise winding — the default in Metal — a triangle is facing us if its vertices are ordered in clockwise fashion from our perspective. The side that is not facing us is called the *back face*.

To override the default front-facing winding of clockwise, we call the `setFrontFacing(_:)` method on our render encoder:

```
renderCommandEncoder.setFrontFacing(.counterClockwise)
```

Now that we have a notion of front and back for triangles, we can save some processing time by telling Metal to ignore triangles that are facing away from the virtual camera. We call this process *back-face culling*.

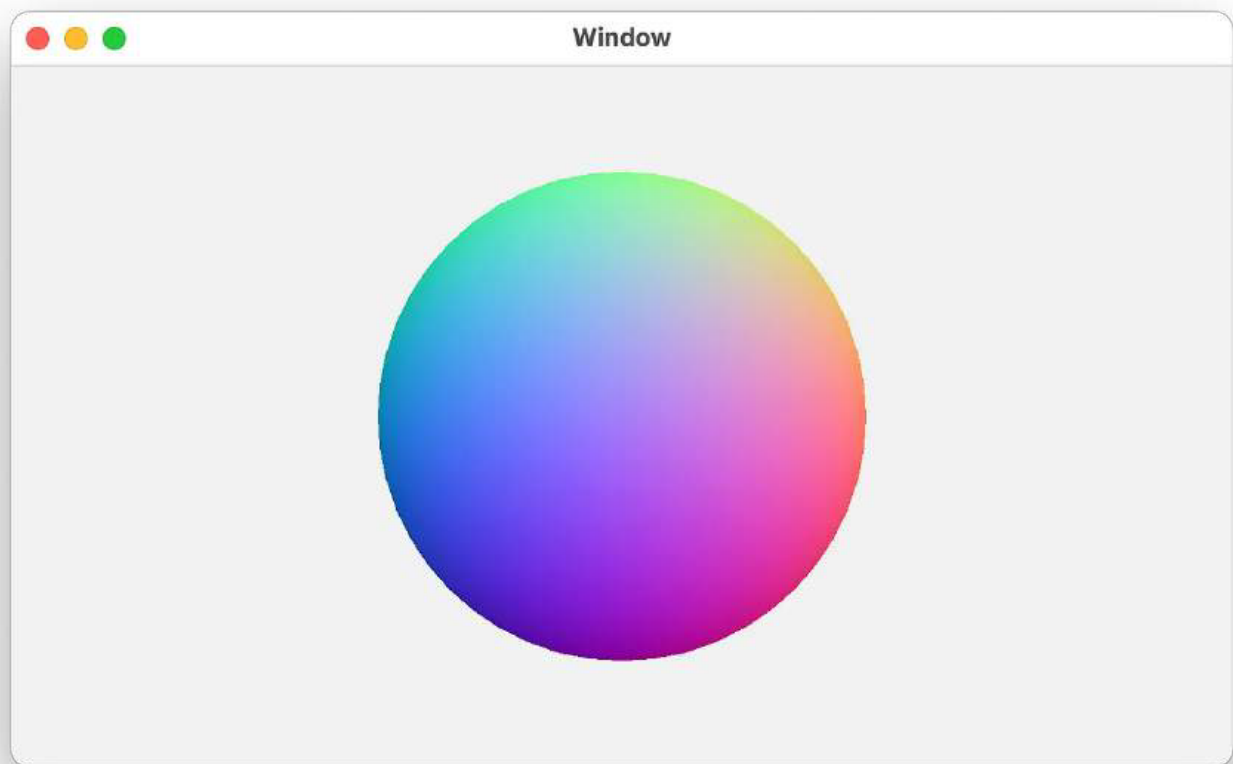
We can tell Metal to cull back faces, cull front faces, or disable culling by calling `setCullMode(_:)` on the render command encoder:

```
renderCommandEncoder.setCullMode(.back)
```

And that's all you need to know to get started using the depth buffer in Metal.

To visualize the 3D surface normals of the sphere, I have replaced the basic lighting fragment function from last time with the following even simpler function:

```
fragment float4 fragment_main(VertexOut in [[stage_in]])
{
    float3 N = normalize(in.normal);
    float3 color = N * float3(0.5) + float3(0.5);
    return float4(color, 1);
}
```



Next time we will introduce perspective projection, an essential tool for creating a more convincing illusion of depth in larger 3D scenes.

# Day 14: Perspective



Warren Moore ·

9 min read · Apr 15, 2022

*This series of posts is my attempt to present the Metal graphics programming framework in small, bite-sized chunks for Swift app developers who haven't done GPU programming before.*

*If you want to work through this series in order, start [here](#).*

In the previous article we entered the third dimension by introducing depth buffering, which allows us to implicitly sort surfaces by their distance.

So far, we've mostly avoided a detailed discussion of the different coordinate spaces we use when going from vertices to pixels. Today, we'll take a closer look at exactly what our transformations are doing and introduce a new type of projection transformation.

## **Model Space**

In previous articles, I have alluded to a mesh's "local origin," which implies that the mesh's vertices are specified relative to some coordinate system.

We call this coordinate system "model space" or "object space."

Usually, the author of the 3D model decides which coordinate system to model in. For objects as well as legged creatures, it often makes sense to place the origin on an imagined ground plane, since many objects spend most of their time resting on flat surfaces. On the other hand, it may be more useful to select the object's center of gravity as its local origin instead.

As for selecting the scale of an object, it is often useful to use real-world units like meters as the units of the model. This allows objects to be

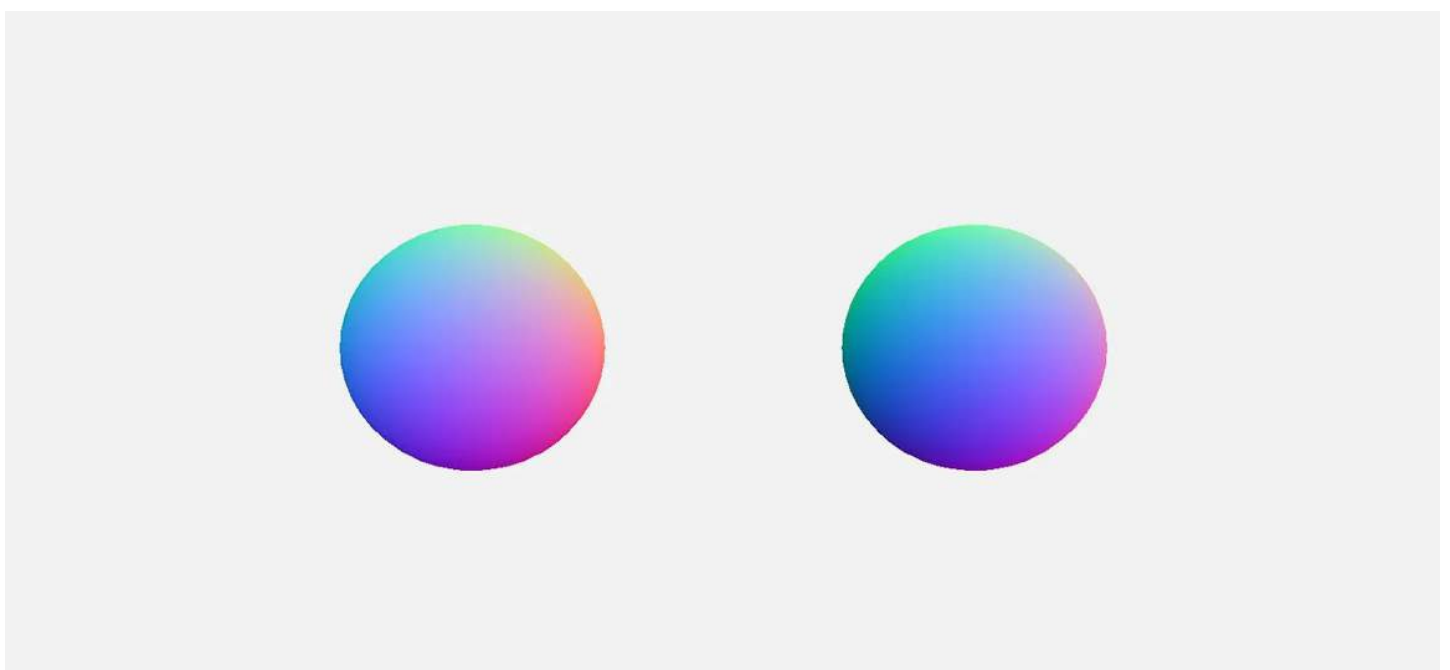
composed together in larger scenes without having to adjust their scales independently. For some applications that involve very large or very small distances, it may make sense to use different units, to avoid issues with numerical precision that can occur when transformations are composed.

Now that we know how vertices are defined in meshes and models, let's talk about how we build virtual worlds by situating objects relative to one another.

## ***World Space***

As its name implies, *world space* is a global coordinate system relative to which objects can be arranged. We move between model space and world space by assigning each object a *model-to-world transformation*, colloquially called a *model transformation* (or *world transformation*), that moves points from model space into world space. We have seen model transformations in action when combining rotations, translations, and scaling in 2D.

As an example of a 3D model transformation, consider placing two spheres together in a scene. One sphere might be given a model transformation that translates it two units to the left, while the other is translated two units to the right.



In model space, the vertices of the two spheres are identical, but each vertex is transformed according to its sphere's model transformation, which

causes the vertices to have different coordinates in world space. In this same way, we can build a scene comprised of many different objects.

## ***View Space***

Now we know how to situation objects relative to one another in a unified, global coordinate space: we give each object its own model transformation.

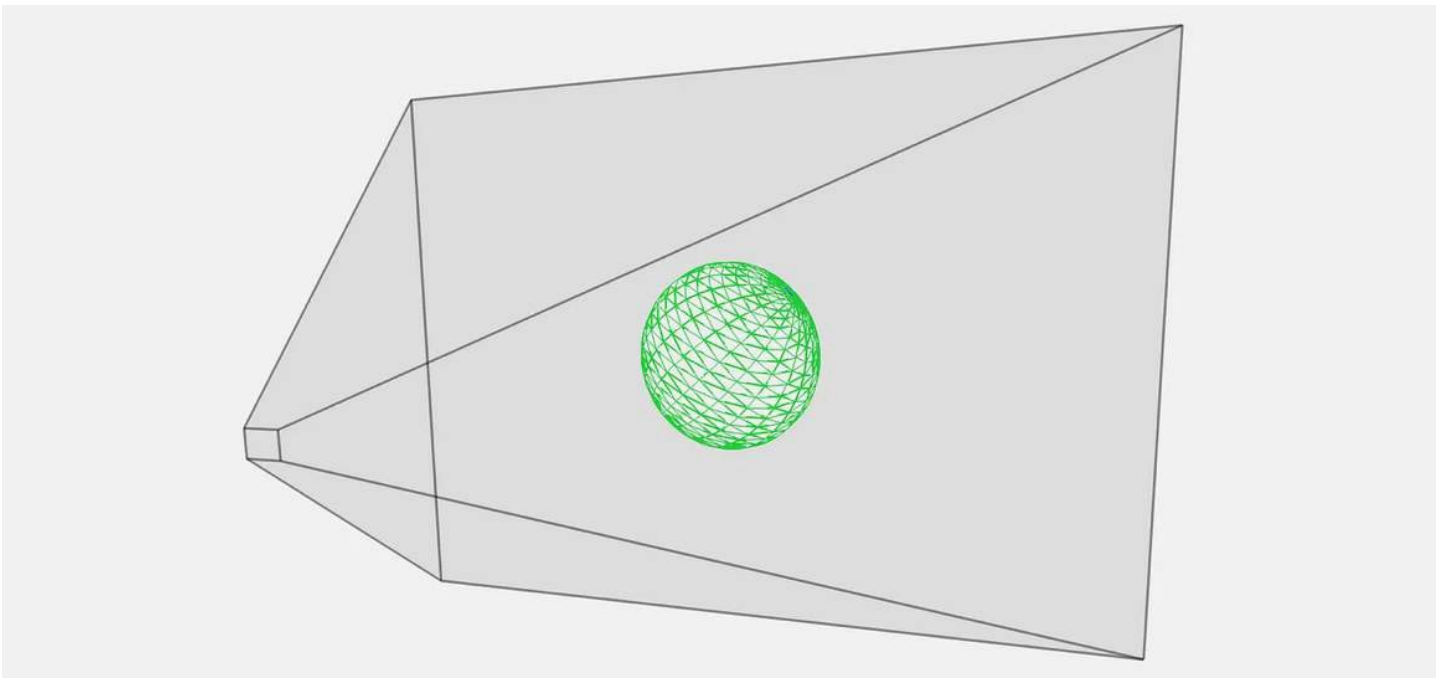
We have mentioned the notion of a “virtual camera” a couple of times without adequately defining it. When we speak of the camera in graphics, we mean the point of view from which we look at the scene, along with other parameters such as the field of view.

The position and orientation of the camera together form a coordinate space called *view space*, *eye space*, or *camera space*. Once the camera is positioned, its orientation can be set by selecting directions that point right (the x axis) and up (the y axis). By the right-hand rule, the z axis then points toward the viewer, *nottoward* the world as you might expect. According to this convention, we view the world along the camera’s negative-z axis.

To find the view transformation, we take the *inverse* of the camera’s model- to-world transformation. To see why, consider how transformations move us between coordinate systems: if a transformation takes us from a coordinate system **A** to a coordinate system **B**, its inverse takes us from **B** back to **A**. Since we want to put the world in front of the camera rather than *vice versa*, we use the inverse as the view transformation.

We refer to the region of space that is visible as the *viewing volume*. Any points not inside the viewing volume will not appear in our rendered image.

Once we introduce perspective, the world-space viewing volume is not a rectangular prism. Instead, it is a shape called a *frustum*, which is like a rectangular pyramid with its top cut off. The pyramid’s primary axis is oriented along the camera’s (negative) z axis, with the apex at the camera’s location. The view frustum is illustrated below.



## That *W* Coordinate

In the same way that 3D points and vectors have 3 components ( $x$ ,  $y$ , and  $z$ ), 4D points and vectors have 4 components:  $x$ ,  $y$ ,  $z$ , and  $w$ . This last component has already been useful to us in unifying the different types of transformations into matrix form, but it has additional uses.

Recall that vectors have the same magnitude and direction regardless of where they are in a coordinate space. This implies that they transform differently from points. Specifically, applying a translation matrix to a vector should have no effect. We can achieve this by setting the vector's  $w$  component to 0, which zeroes out the translational component of a transformation matrix.

On the other hand, we need points to be able to translate, so we set their  $w$  component to 1. This gives us the 3D translation results we want.

What about other values of  $w$ ? Is there ever a time when we want values between 0 and 1, or perhaps greater than 1 or less than zero?

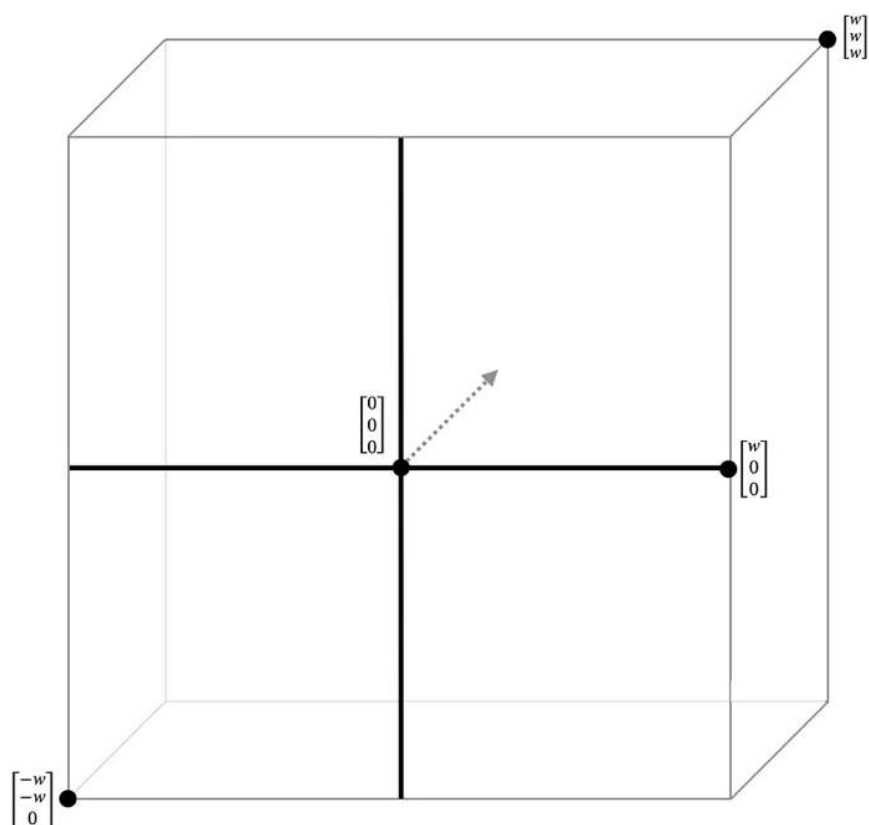
## Clip Space

When working in two dimensions, we frequently said that we were working in NDC. This is only partially true. The true job of the vertex function is to produce positions in *clip space*.

So what is clip space? Clip space is a coordinate space in which the viewing volume is bounded along the x and y axes between  $-w$  to  $w$  and the z axis between 0 to w. What does it mean for a space to be bounded in this way?

As we will discuss below, the projection matrix calculates the w coordinate of the output position different depending on the intended effect of the projection. This means that some projection matrices will produce w values that are not 0 or 1. As part of the vertex processing pipeline, vertices' x, y, and z coordinates are compared to their w coordinate to determine whether they are inside the viewing volume.

The figure below illustrates clip space, with several points labeled.



This method of determining whether a vertex is “in” or “out” is what gives clip space its name: primitives are “clipped” against the boundary of the viewing volume on their way to rasterization.

One quirk of clip space is that it is “left-handed”: the z axis points away from the viewer. This contrasts with our chosen convention for model space, world space, and view space, which are all right-handed. This implies that we need to flip the z axis when moving from view space to clip space.

Indeed, we did this implicitly with the orthogonal projection matrix we





introduced on day 9. It is necessary to remember this difference when formulating other projection matrices.

## ***The Perspective Divide***

The job of the vertex function is to produce positions in clip space. In addition to scaling and translating points from their view-space positions into the appropriate ranges, the projection transform also does something else: it sets up the  $w$  coordinate for the *perspective divide*.

Just after the vertex function runs, before rasterization, the  $x$ ,  $y$ , and  $z$  components of each vertex are divided by the  $w$  coordinate. Why would we do such a thing? The primary reason is to introduce perspective, hence this process is called the perspective divide.

Due to a phenomenon called convergence, the human vision system perceives distant objects to be closer to the center of the field of vision. One example of this phenomenon is how train tracks appear to converge in the distance.

We achieve perspective in 3D rendering by deriving the  $w$  coordinate from the  $z$  coordinate as part of our projection matrix in the vertex shader. Then, when the perspective divide occurs, more distant  $z$  values — which are larger in magnitude — cause  $x$  and  $y$  to shrink proportionately, producing convergence and foreshortening (the illusion that parts of objects that are farther away are smaller).

Our orthogonal projection matrix never produced points whose  $w$  components were anything other than 1. This is because an orthogonal projection is a *parallel projection*: parallel lines remain parallel after projection rather than converging. If we actually want perspective, we need a new kind of projection matrix.

## ***Perspective Projection***

There are numerous ways to parameterize a projection matrix. One very popular set of parameters is: near plane distance, and far plane distance,

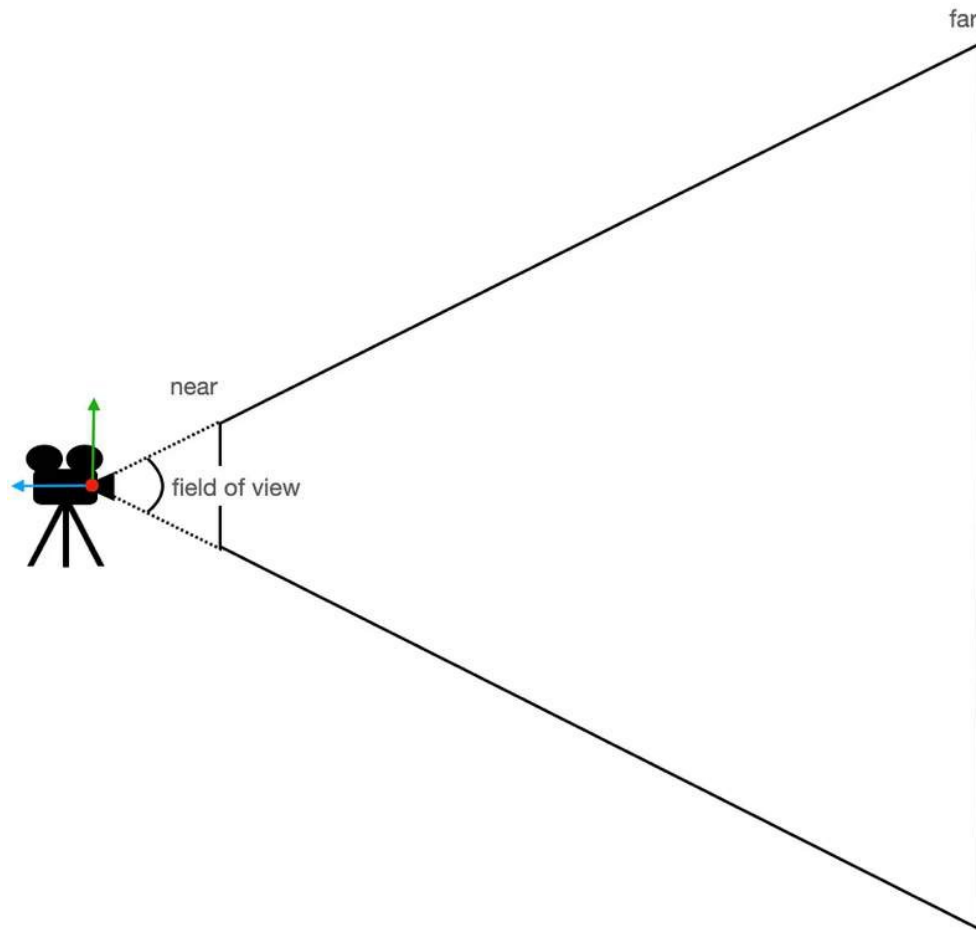
aspect ratio, and field of view.

We have already used near and far z values to delineate our view-space viewing volume when performing orthographic projection. In the same way, when building a projection matrix, we select near and far z values to define the distance to the near plane and far plane. By convention these values are positive, but because the z axis of view space points toward the viewer, they actually represent distances along the negative z axis. This is accounted for in the projection matrix math.

If our rendered image is square, we do not need to account for the aspect ratio, since the image plane is also a square in normalized device coordinates. However, since we will often be displaying our rendered images in a non-square view, we need to adjust our projection matrix accordingly.

We also need to select a field of view angle to determine how much of the virtual world is visible. We can select either a horizontal field of view or vertical field of view, then compute the other based on the aspect ratio.

Here, we will choose a vertical field of view, which is the angular measure between the top and bottom planes of the view frustum. Larger angles capture more of the scene, but using extremely large angles produces non-physical distortion. The goal should be to choose a sensible angle that accounts for the context of the virtual camera, the dimensions of the rendered image, and the expected viewing distance of the user.

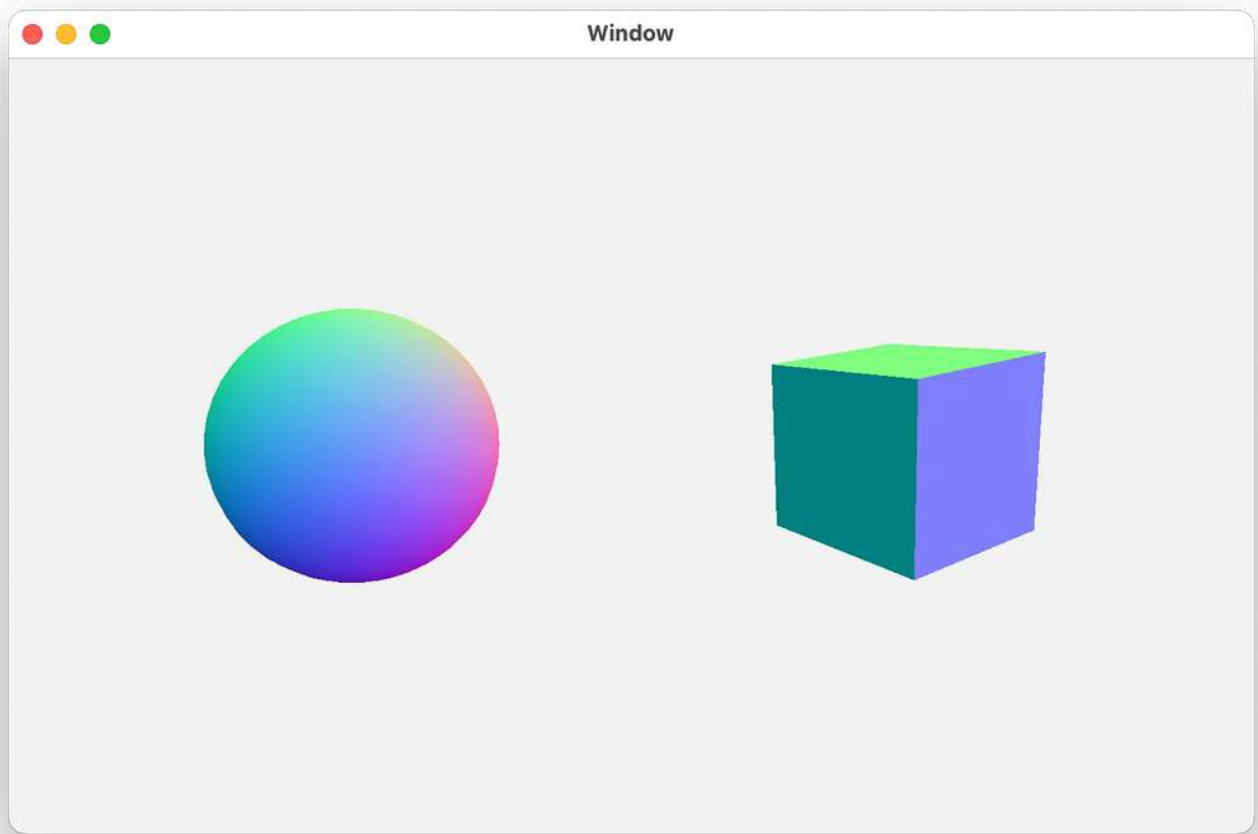


We will not go through the complete derivation of perspective projection here. If you are curious about the theory, I recommend [Song Ho Ahn's excellent article](#) on the subject. For now, here is a Swift function that generates a projection matrix given the parameters above:

```
init(perspectiveProjectionFoVY fovyRadians: Float,
    aspectRatio: Float,
    near: Float,
    far: Float)
{
    let sy = 1 / tan(fovyRadians * 0.5)
    let sx = sy / aspectRatio
    let zRange = far - near
    let sz = -(far + near) / zRange
    let tz = -2 * far * near / zRange
    self.init(SIMD4<Float>(sx, 0, 0, 0),
              SIMD4<Float>(0, sy, 0, 0),
              SIMD4<Float>(0, 0, sz, -1),
              SIMD4<Float>(0, 0, tz, 0))
}
```

Replacing our existing projection matrix with a perspective projection matrix enables us to produce images that have more realistic proportions. Check out the [sample code](#) to see how to add support for multiple objects and introduce a view transformation that positions the virtual camera somewhere other than

the origin.



We will discuss these changes and learn about how to create hierarchical arrangements of objects next time.

# Day 15: Hierarchy



Warren Moore ·

9 min read · Apr 16, 2022

*This series of posts is my attempt to present the Metal graphics programming framework in small, bite-sized chunks for Swift app developers who haven't done GPU programming before.*

*If you want to work through this series in order, start [here](#). To download the sample code for this article, go [here](#).*

In the last article, we introduced the perspective projection matrix, which allows us to account for phenomena of the human visual system such as foreshortening and convergence.

We also briefly touched on the idea of using different transforms with the same mesh to render multiple objects in the same frame. In this article, we will further explore how to create *hierarchies* of objects: groups of objects that move relative to one another.

## **Representing Hierarchy: A Node Class**

Sometimes, we want one object to move relative to another. A classic example of this is a robot arm.



Photo by [Possessed Photography](#) on [Unsplash](#)

The hand or actuators are connected to the “forearm” joint, which is connected to the “shoulder” joint. When the shoulder rotates, the other parts of the arm move relative to it.

Another example is planetary and lunar orbits. Because a planet is gravitationally bound to its star, the planet revolves around the star relative to the star’s own motion, and moons likewise follow their planets as they revolve.

We can model this in our graphics applications by introducing the concept of *hierarchical transformations*. If one object always moves relative to another object, we call the first object the child and the second object the parent.

To model hierarchy, we will introduce a new class named `Node`. A node combines an optional mesh with a transformation matrix, and represents a discrete object in a 3D scene. We’ll also give our node a color for demonstration purposes:

```
class Node {
    var mesh: MTKMesh?
    var color = SIMD4<Float>(1, 1, 1, 1)
    var transform: simd_float4x4 = matrix_identity_float4x4
```

To allow nodes to be parented by other nodes, we will maintain an array of child nodes, along with a weak reference to the parent of the node, if it exists.

```
weak var parentNode: Node?
private(set) var childNodes = [Node]()
```

Since we need to manage the bidirectional relationships between parent and child nodes, we make the setter of the `childNodes` property private and offer methods that allow adding and removing of child nodes:

```
func addChildNode(_ node: Node)
{
    childNodes.append(node)
    node.parentNode = self
}

func removeFromParent()
{
    parentNode?.removeChildNode(self)
}

private func removeChildNode(_ node: Node)
{
    childNodes.removeAll { $0 === node }
}
}
```

Finally, since the purpose of these parent-child relationships is to allow children move relative to parents, we add a `worldTransform` computed property that composes (multiplies) the node's parent's transformation matrix with its own. This produces a matrix that accumulates transforms up to the root node, so each child moves relative to all of its ancestors in the hierarchy.

```
var worldTransform: simd_float4x4
{
    if let parent = parentNode {
        return parent.worldTransform * transform
    } else {
        return transform
    }
}
```

If a node has no parent, it is considered a root node, and its model-to-world



transformation is just its local transformation.

Now that we have a concept of hierarchy, let's talk about how we render multiple distinct objects.

## ***Bundling Constant Data***

We want each object to have its own color, but how do we get this color into our shader functions so we can use it to tint our objects?

Up until now, we have been passing a single matrix into our vertex shader so we can transform our vertices into clip space. Instead of allocating another buffer just to hold colors, we can create a struct that groups together all of the constant data for a node, called `NodeConstants`. In Swift, it looks like this:

```
struct NodeConstants {  
    var modelViewProjectionMatrix: float4x4  
    var color: SIMD4<Float>  
}
```

The corresponding definition in Metal Shading Language is this:

```
struct NodeConstants {  
    float4x4 modelViewProjectionMatrix;  
    float4 color;  
};
```

Whenever dealing with vector types, we always need to be mindful of alignment. In this case, the alignment of both `float4x4` and `float4` is 16 bytes, so these structures don't need to be padded to be correctly aligned in memory.

We now update our vertex function signature to take a reference to our new constants struct:

```
vertex VertexOut vertex_main(  
    VertexIn in [[stage_in]],  
    constant NodeConstants &constants [[buffer(2)]])
```

We also augment our vertex function's output structure with a color member so it can carry the node's color into the fragment shader:

```
struct VertexOut {  
    float4 position [[position]];  
    float3 normal;  
    float4 color;  
};
```

Later, when rendering, we will see how to write node constants into our dynamic constants buffer.

## ***Building a Hierarchy of Nodes***

To demonstrate hierarchical transformation, we will build a small solar system with a sun, planet, and moon. The moon will revolve around the planet, and the planet will revolve around the sun.

We will reuse the sphere mesh we created last time to act as the visual representation of each orb.

Rather than having our renderer hold onto a mesh directly, we will replace the `mesh` member with members that hold our set of nodes:

```
var sunNode: Node!  
var planetNode: Node!  
var moonNode: Node!  
var nodes = [Node]()
```

The reason we store references to each node and also keep them in an array is so we can refer to the nodes individually when animating their transformations.

We create each node by instantiating it with the sphere mesh and supplying its color:

```
sunNode = Node(mesh: sphereMesh)
sunNode.color = SIMD4<Float>(1, 1, 0, 1)

planetNode = Node(mesh: sphereMesh)
planetNode.color = SIMD4<Float>(0, 0.4, 0.9, 1)

moonNode = Node(mesh: sphereMesh)
moonNode.color = SIMD4<Float>(0.7, 0.7, 0.7, 1)
```

We then create the relationships among our nodes with the `addChildNode` method:

```
sunNode.addChildNode(planetNode)
planetNode.addChildNode(moonNode)
```

Finally, we store a list of nodes so we can easily iterate over them when updating constants and rendering:

```
nodes = [sunNode, planetNode, moonNode]
```

## ***Allocating Space for Constants***

We will continue using the same dynamic constant buffer update scheme we introduced on day 9, but we need to make our buffer bigger to accommodate multiple objects.

We can define a global variable that contains the maximum number of objects in a scene. If we ever need more, we can just update this number and recompile.

```
let MaxObjectCount = 16
```

We also update the size of our per-draw call constants, since we are now passing a `NodeConstants` struct instead of a plain matrix:

```
self.constantsSize = MemoryLayout<NodeConstants>.size
```

We need to allocate a constants buffer that is able to store a copy of node constants for every object for as many frames as we might have in-flight at once:

```
let constantBufferLength = constantsStride * MaxObjectCount *
MaxOutstandingFrameCount

constantBuffer =
    device.makeBuffer( length:
        constantBufferLength,
        options: .storageModeShared)
```

We assign each node a unique index — its index in the `nodes` array. Then we can write a method that calculates the appropriate offset for the node's constants in a given frame:

```
func constantBufferOffset(objectIndex: Int, frameIndex: Int) -> Int
{
    let frameConstantOffset =
        (frameIndex % MaxOutstandingFrameCount) *
        MaxObjectCount * constantsStride

    let objectConstantOffset = frameConstantOffset +
        (objectIndex * constantsStride)

    return objectConstantOffset
}
```

Now let's talk about the changes we'll make to the `updateConstants()` method to get our solar system in motion.

## ***Computing the View Matrix***

Recall from the previous article that we introduced a new matrix that is composed in-between our model-to-world matrix and our projection matrix:

the *view matrix*. The purpose of the view matrix is to allow us to

move the virtual camera around the scene without having to change the transformations of any of our objects.

Our simple camera will sit 5 units from the origin along the positive z axis and will be oriented to point straight down the negative z axis. This means that its transformation is just a translation matrix. We write down the camera's position in world space, then invert the transformation by moving in the opposite direction:

```
let cameraPosition = SIMD3<Float>(0, 0, 5)
let viewMatrix = simd_float4x4(translate: -cameraPosition)
```

This has the effect of moving the world 5 units along the negative z axis, away from our point of view.

## ***Computing Transformation for Orbital Dynamics***

Since our node hierarchy takes care of relative motion, we only need to think about how each node moves relative to its own parent. The sun is a root node, and doesn't move at all. The planet revolves around the sun, and the moon revolves around the planet. The planet and the moon will also be scaled to produce a difference in size among the various orbs.

The scaling transformation is applied first:

```
let planetRadius: Float = 0.3
let scale = simd_float4x4(scale:
    SIMD3<Float>(repeating: planetRadius))
```

This makes the planet 30% of the size of the sun.

Next, we need to translate the planet away from the sun by its orbital radius. We do this with a translation matrix:

```
let planetOrbitalRadius: Float = 2
```



```
let translate = simd_float4x4(translate:
    SIMD3<Float>(planetOrbitalRadius, 0, 0))
```

Finally, to make the planet revolve, we rotate this translated coordinate system around the sun’s “up” axis, which is just the global y axis, `[0, 1, 0]`.

```
let yAxis = SIMD3<Float>(0, 1, 0)
let rotate = simd_float4x4(rotateAbout: yAxis, byAngle: t)
```

Using the time variable `t` as our rotation angle causes the planet’s revolution to animate over time.

We compose the planet’s complete transformation by multiplying these matrices together. The effect of each matrix applies to a vertex in left-to-right order: *first scale, then translation, then rotation*.

```
planetNode.transform = rotate * translate * scale
```

Note that the rotation and translation are reversed from the order we previously used when drawing in 2D. This is because we want the planet to revolve around the center of the solar system rather than rotating around its own center.

We could add another rotation matrix in between the scale and translation if we also wanted the planet to rotate on its own axis, but we omit that effect here.

## ***Updating Dynamic Constants***

Since each node has a unique model-to-world transformation, we will compute the complete transform matrix that we send into the vertex shader for each node.

We will iterate over our node list, writing each node’s constants into the



constant buffer.

```
for (objectIndex, node) in nodes.enumerated() {
```

We compose the node's model matrix with the view matrix and projection matrix to produce a combined *model-view-projection matrix*, which takes us from model space all the way to clip space with a single multiplication in the vertex shader.

```
let transform = projectionMatrix * viewMatrix * node.worldTransform
var constants = NodeConstants(
    modelViewProjectionMatrix: transform,
    color: node.color)
```

Note that we use the node's computed `worldTransform` property to combine its own transform with that of its parents: this is what produces the hierarchical transformation effect.

We compute the node's constant buffer offset by using the

```
constantBufferOffset(objectIndex:frameIndex:) method we wrote earlier:
```

```
let offset = constantBufferOffset(objectIndex: objectIndex,
                                   frameIndex: frameIndex)
```

And finally we write the node constants into the buffer:

```
let constantsPointer = constantBuffer.contents().advanced(by:
offset)

constantsPointer.copyMemory(from: &constants, byteCount:
constantsSize)
```

Now let's update our shaders and issue some draw calls.

## Updating the Shader Functions

Not much needs to change in the vertex shader: we multiply our model- space vertex positions by the unified model-view-projection matrix and pass the node's color through:

```
vertex VertexOut vertex_main(
    VertexIn in [[stage_in]],
    constant NodeConstants &constants [[buffer(2)]]
)
{
    VertexOut out;
    out.position = constants.modelViewProjectionMatrix *
float4(in.position, 1.0);
    out.normal = in.normal;
    out.color = constants.color;
    return out;
}
```

We will use a trivially simple fragment function to demonstrate that passing through the node's color works:

```
fragment float4 fragment_main(VertexOut in [[stage_in]])
{ return in.color;
}
```

## Updating the Draw Method

Very little has to change in the draw method. We need to iterate over each node and encode a draw call for each node that has a mesh, using the appropriate constant buffer offset:

```
for (objectIndex, node) in nodes.enumerated()
{ guard let mesh = node.mesh else { continue }

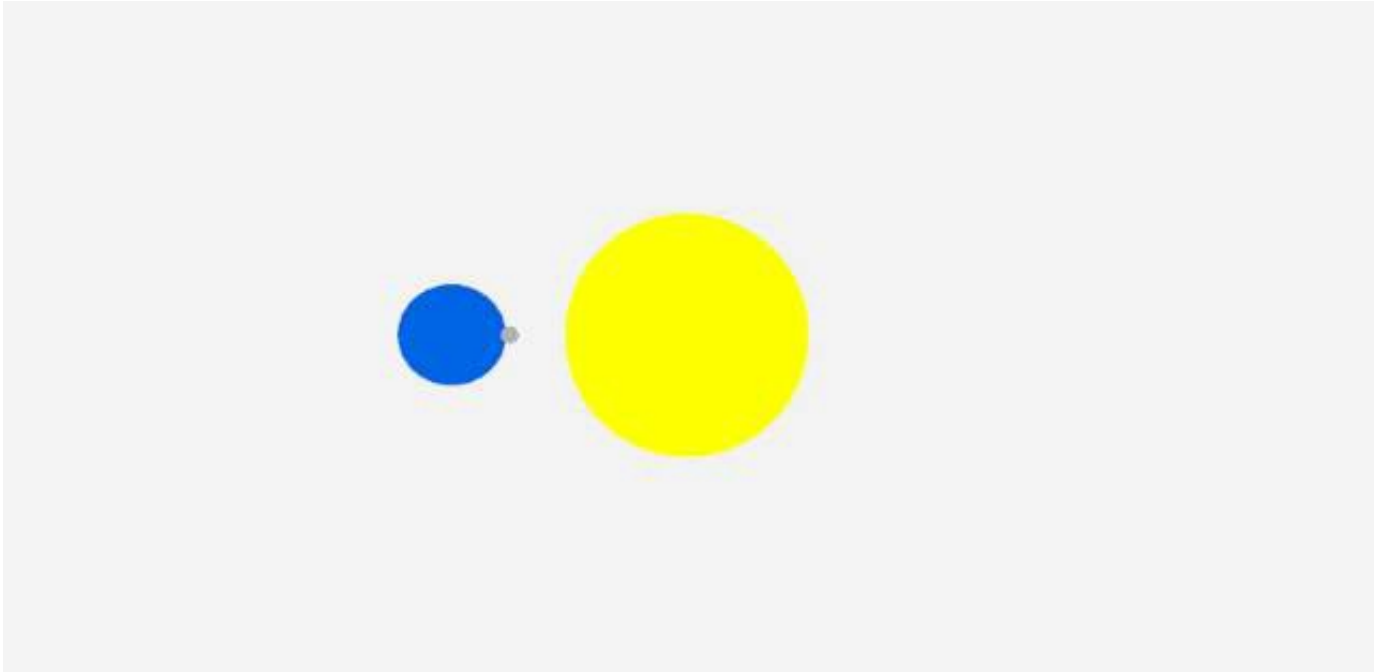
    let offset =
        constantBufferOffset( objectIndex: objectIndex, frameIndex:
                                frameIndex)

    renderCommandEncoder.setVertexBuffer(constantBuffer,
                                          offset: offset,
                                          index: 2)

    // Bind vertex buffers, etc.
    // Encode draw call
}
```



With these updates made, we can run the sample app and see our little solar system in motion:



Next time, we will finally get introduced to another resource type: textures.

# Day 16: Textures



Warren Moore ·

11 min read · Apr 17, 2022

*This series of posts is my attempt to present the Metal graphics programming framework in small, bite-sized chunks for Swift app developers who haven't done GPU programming before.*

*If you want to work through this series in order, start [here](#). To download the sample code for this article, go [here](#).*

So far, we have only worked with one type of resource: buffers. Although buffers are extremely versatile, sometimes we need a little more structure. In particular, sometimes we want to store images on the GPU. To meet this need, we'll introduce a new type of resource: textures.

We have already been using textures behind the scenes: the drawables provided by our `MTKView` wrap color textures, and enabling depth buffering on the view causes it to allocate a depth texture for us. In this article, we will learn how to create our own textures, and how to apply them to objects in our scene to give them more apparent detail.

NOTE: When referring to a pixel in a texture, we sometimes use the more specific term *texel* to emphasize where it resides. I will only employ this term when it is important to distinguish between pixels and texels; otherwise, I prefer “pixel.”

## Textures in Metal

Like a buffer, a texture owns a region of GPU memory. Unlike a buffer, a texture has an innate *pixel format*, which indicates how the data it contains should be interpreted. For example, the `MTLPixelFormat.bgra8Unorm` format

tells us that each pixel has four components: *blue*, *green*, *red*, and *alpha*; that each of these components occupies *8 bits*; and that each component should be interpreted by taking its 8-bit *unsigned* integer value (between 0 and 255) and dividing it by 255 to get a *normalized* value between 0 and 1. That’s a lot of information in a single property.

In Metal, textures can be one-dimensional, two-dimensional, or three-dimensional. Additionally, a texture can store a *cube map*, which is a set of six images that are stitched together to form a cube. Finally, a texture can be a *texture array* of any of the previous types (1D, 2D, 3D, or cube), which means it can effectively store a list of textures as a single resource.

Like buffers, textures can be stored differently depending on how often they will be accessed by the CPU and the GPU. This is determined by the texture’s *storage mode*. On macOS, a texture with a storage mode of

`MTLStorageMode.managed` can be read and written by the CPU without explicit copies, but explicit synchronization is required. A texture with a storage mode of

`MTLStorageMode.private` can only be read and written by the GPU, so updating its contents from the GPU requires creating a “staging resource,” uploading that resource to the GPU, then using a blit command encoder to copy into the private texture. Finally, a texture with a storage mode of

`MTLStorageMode.shared` on iOS or Macs with Apple Silicon can be read and written by the CPU and GPU without using Metal’s explicit synchronization APIs, although it is still important to pay attention to ordinary synchronization concerns like race conditions.

One more consideration when creating textures is how they will be used. Some textures, like render targets, will be written frequently, but rarely if ever read; their usage flags should include `MTLTextureUsage.renderTarget`. By contrast, many textures, like those used to perform texture mapping, will be read very frequently and rarely if ever updated; their usage flags should include

`MTLTextureUsage.shaderRead`. By telling Metal how we expect to use our textures, we allow the driver to optimize the texture’s layout and access patterns.

## ***Creating a Texture***

To create a texture, we first fill out an `MTLTextureDescriptor` object. This parameter object has many properties that control the format and layout of the texture.

```
let textureDescriptor = MTLTextureDescriptor()
```

Suppose we want to create a texture that we will populate with image data once from the CPU, then sample from repeatedly when rendering. We can start by setting the pixel format:

```
textureDescriptor.pixelFormat = MTLPixelFormat.bgra8Unorm
```

Then, we configure the texture type and size by setting the `width` and `height` properties. There is also a `depth` property for setting the depth of 3D textures; this is set to 1 by default.

```
textureDescriptor.textureType = MTLTextureType.type2D
textureDescriptor.width = 128
textureDescriptor.height = 128
```

We set its usage flags to `.shaderRead`:

```
textureDescriptor.usage = MTLTextureUsage.shaderRead
```

Finally, we set the texture's storage mode to `.private`, since it will be used exclusively by the GPU.

```
textureDescriptor.storageMode = MTLStorageMode.private
```



Now that we have our texture descriptor populated, we create the texture by

calling the `makeTexture(descriptor:)` method on the device:

```
let texture = device.makeTexture(descriptor: textureDescriptor)!
```

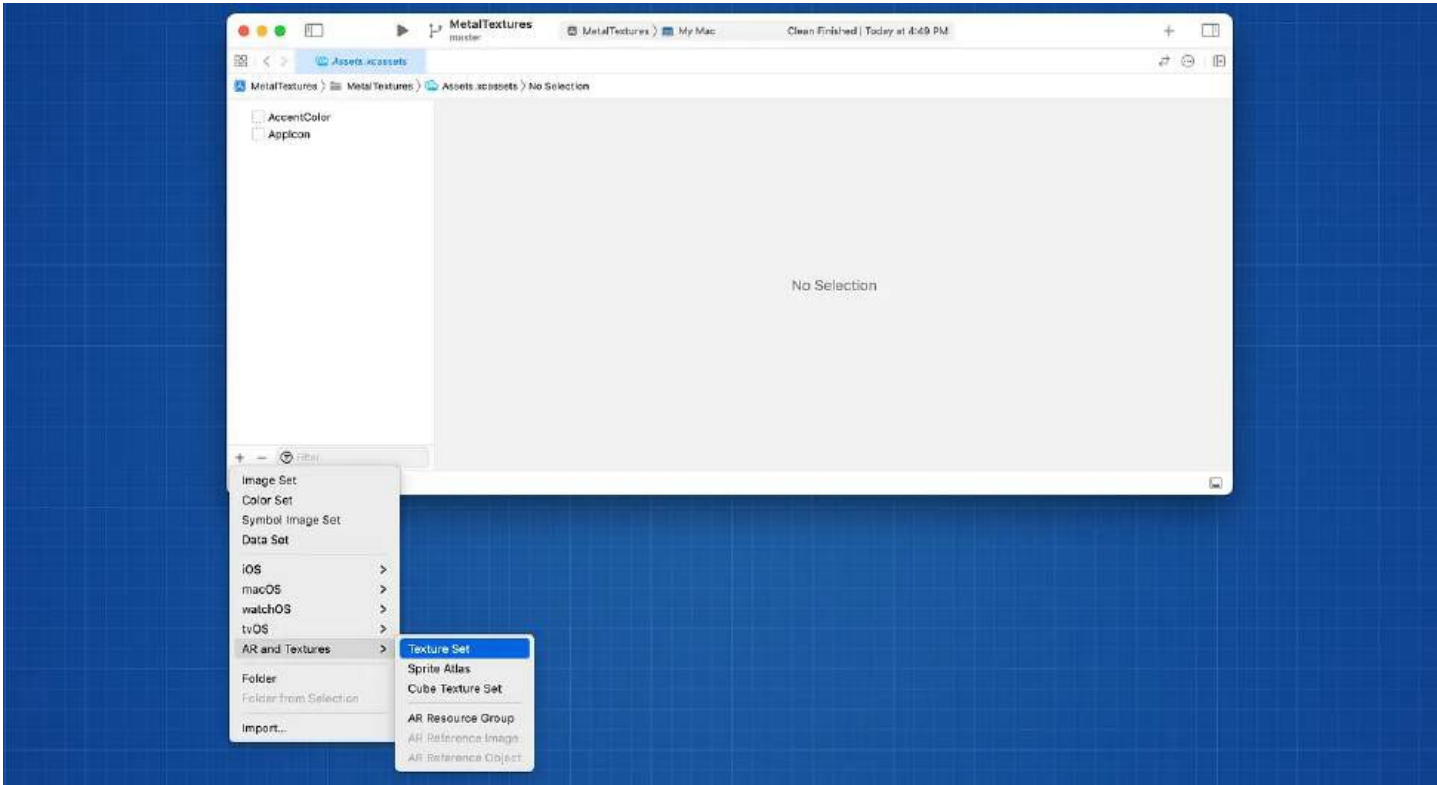
Now the texture is allocated but has no contents. To fill it with image data, we would load an image into memory, copy the data into a buffer, then use a blit command encoder to populate the texture.

Fortunately, it is commonly easier to create and populate textures using a utility class from MetalKit called `MTKTextureLoader`.

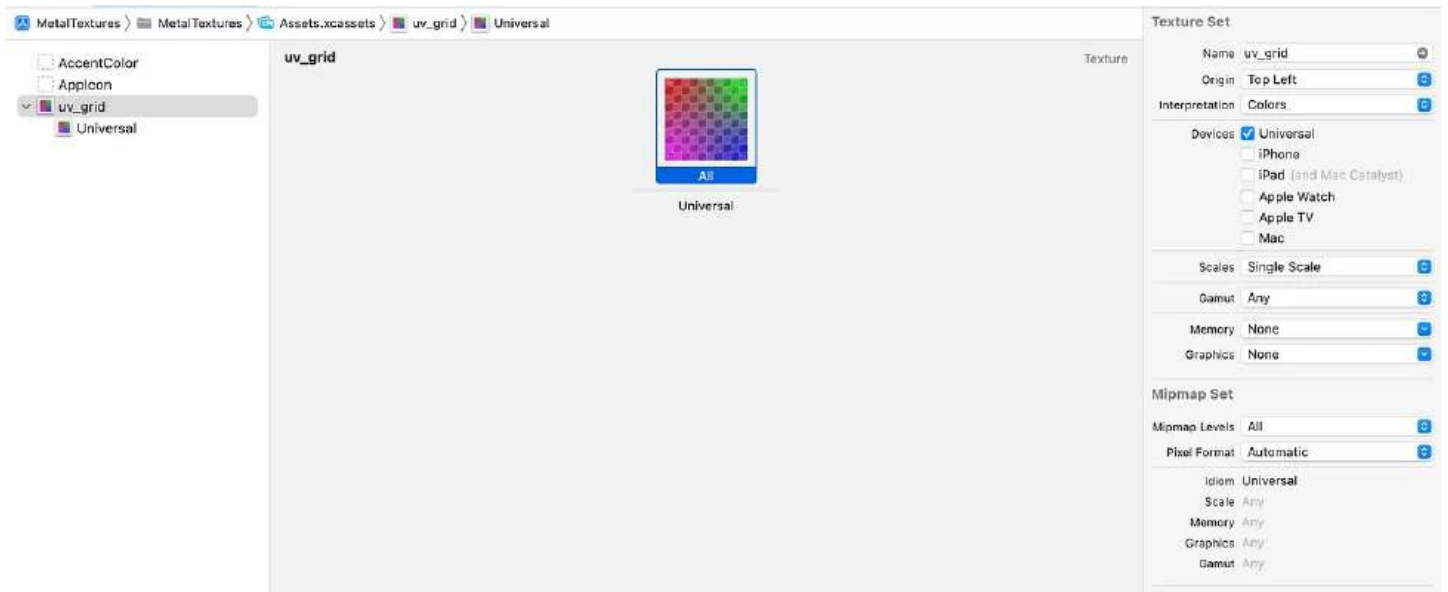
## Loading a Texture from an Asset Catalog

Asset catalogs are a powerful way to organize app resources efficiently. You have probably used them to manage your app’s icon assets, and perhaps have used them much more extensively. We will look at how to use Asset catalogs and MetalKit together to easily and efficiently create Metal textures at runtime.

To start, add a new Texture Set asset to your app’s asset catalog.



Then, drop an image file onto the image well in the asset’s overview:



You can specify different asset representations that distinguish by traits like device type, scale, and graphics family, but the default Universal representation will suffice for our needs. Make sure to set its “Origin” property to “Bottom Left”; otherwise, it will be loaded into Metal upside- down.

We can load texture assets with an instance of `MTKTextureLoader`. We create a texture loader by giving it a device with which it can allocate resources and do work on our behalf:

```
let textureLoader = MTKTextureLoader(device: device)
```

To specify texture properties that cannot be encoded in a resource catalog, we create a dictionary of `MTKTextureLoader.Options`. We want to specify the texture’s storage mode and usage flags so Metal knows where to allocate the texture and how it will be used:

```
let options: [MTKTextureLoader.Option : Any] = [
    .textureUsage : MTLTextureUsage.shaderRead.rawValue,
    .textureStorageMode : MTLStorageMode.private.rawValue
]
```

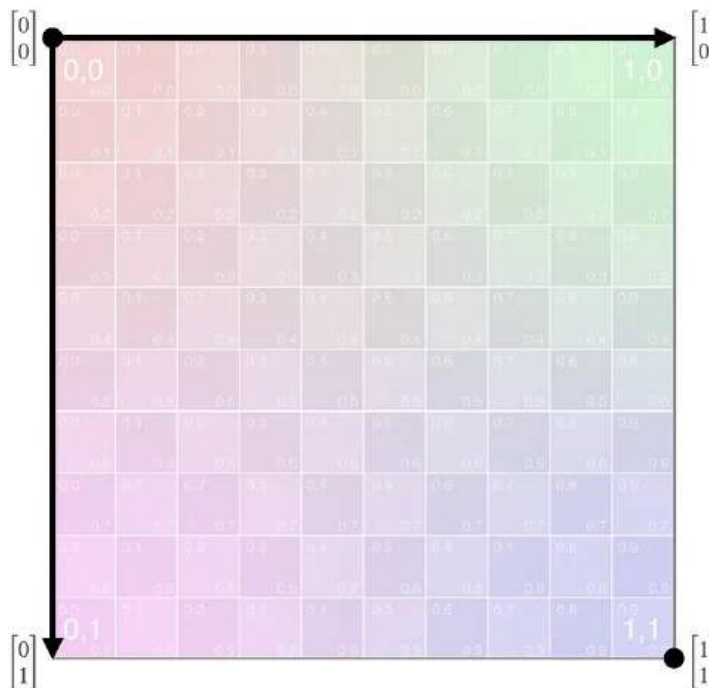
Then, to load the texture, we provide its name and our options dictionary to the loader:

```
texture = try? textureLoader.newTexture(name: "uv_grid",
                                         scaleFactor: 1.0,
                                         bundle: nil,
                                         options: options)
```

This operation can throw, so all of the usual caveats about error checking apply; we’re using `try?` for the sake of simplicity.

## Texture Space and Texture Coordinates

In Metal, the origin (0, 0) of a texture is in the upper-left corner, with x increasing to the right and y increasing downward. We specify a location within a texture with an (x, y) pair of *texture coordinates*. Texture coordinates are normalized, meaning that x and y span from 0 to 1 regardless of the width and height of the texture.



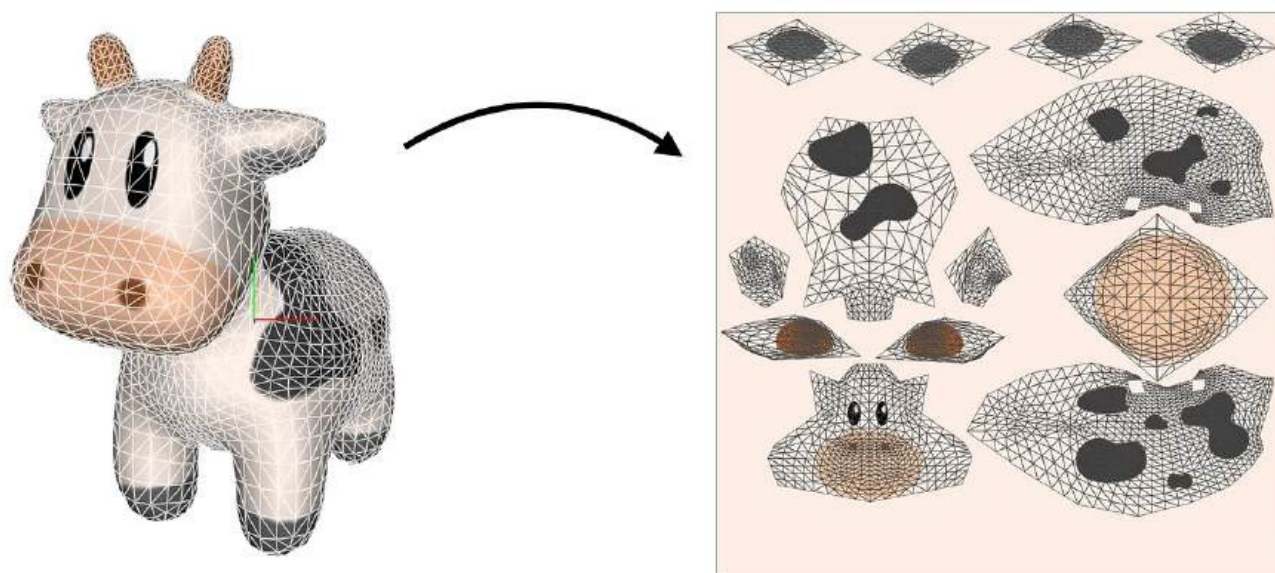
There are various names for these coordinates. For example, you may hear x and y coordinates in texture space called u and v, hence “uv coordinates.” Metal prefers a different convention. In Metal, the horizontal axis is labeled “s,” the vertical axis is labeled “t,” and the depth axis, when used, is labeled “r.”

## Texture Mapping

Texture mapping is one of the most common uses of textures. The purpose

of texture mapping is to introduce more detail to a rendered surface than can be achieved with per-vertex attributes like vertex colors. Since vertex colors are interpolated by the rasterizer, we don't have a chance to provide more detailed color information in between vertices. Texture mapping is one solution to this issue.

Texture mapping is done by “unwrapping” a three-dimensional mesh into one or more contiguous two-dimensional “islands” in texture space. This process is illustrated below.



To store this mapping, each vertex includes a two-element vector attribute that holds its texture coordinates. Like other attributes, texture coordinates are interpolated by the rasterizer, so the fragment function receives a pair of smoothly interpolated texture coordinates that can be used to look up

## ***Generating Texture Coordinates with Model I/O***

We can ask Model I/O to generate texture coordinates when creating an `MDLMesh`. This is as simple as configuring another attribute on the Model I/O vertex descriptor and giving it the name

`MDLVertexAttributeTextureCoordinate`. We also set its offset and buffer index appropriately and update the layout of the buffer itself to account for the new data.

```
mdlVertexDescriptor.vertexAttributes[2].name =  
    MDLVertexAttributeTextureCoordinate  
mdlVertexDescriptor.vertexAttributes[2].format = .float2  
mdlVertexDescriptor.vertexAttributes[2].offset = 24  
mdlVertexDescriptor.vertexAttributes[2].bufferIndex = 0  
mdlVertexDescriptor.bufferLayouts[0].stride = 32
```

So we can give each object a different texture if desired, we replace the

`color` property of the `Node` class with a `texture` property:

```
var texture: MTLTexture?
```

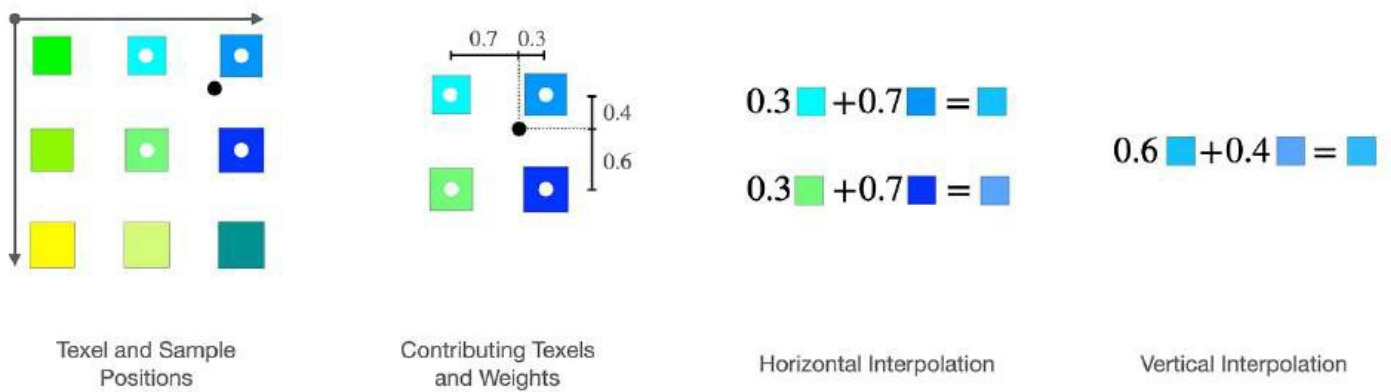
## ***Sampling***

Looking up the color of a texel at a set of absolute coordinates is called *reading*. We might read a texture's contents when writing a per-pixel image filter, or doing some kind of other operation on a pixel-by-pixel basis.

Because they are normalized and interpolated, texture coordinates often indicate positions “in between” texels, so we can't just read a single pixel to get the right color of a fragment; we need to look at multiple nearby texels.

We call the process of calculating the color of a texture at a particular set of texture coordinates *sampling*. It should more accurately be called resampling or reconstruction: textures are already discrete signals, so what we call sampling is really the process of rebuilding the intermediate image data.

Anyway, one of the most common ways to reconstruct a color in between texels is *bilinear interpolation*. In this scheme, the four texels nearest the sample point are considered. The relative distances of the sample point to the center of each texel along the horizontal axis are found and used as weights to produce an interpolated color for the top and bottom pairs of colors, then the relative distances along the vertical axis are used as weights to average these together to get the final color.



Bilinear interpolation is common, but it is not the only possible sampling scheme. We can also ask Metal to round the texture coordinates to the nearest texel center.

These options are available in Metal in the `MTLSamplerMinMagFilter` enumeration:

```
enum MTLMinMagFilter : UInt
{
    case nearest
    case linear
}
```

We will see below where we can use this enumeration to control sampling.

## Address Modes

So far, we have only considered texture coordinates inside the normalized range of 0 to 1. What happens if we sample outside this range? We can choose among several *address modes* to control Metal’s behavior in this case.

Perhaps the most common address mode is *repeat*. This causes the texture to “tile” in texture space, wrapping texture coordinates greater than 1 back around to 0. Another useful option is *clamp*. This forces coordinates less than 0 to 0 and coordinates greater than 1 to 1, causing just the edge texels to repeat outside the texture’s bounds. There are also a few other less common modes; you can consult the Metal documentation for the

`MTLSamplerAddressMode` enumeration for these.

```
enum MTLAddressMode : UInt {
```





```
case clampToEdge = 0
case repeat = 2
//...other modes...
}
```

## ***Sampler States***

We control how textures are sampled by creating *sampler state* objects. Similar to a depth-stencil state, a sampler state contains numerous properties that we can set all at once by binding the sampler on a render command encoder before drawing.

To create a sampler state, we first configure an `MTLSamplerDescriptor` object.

```
let samplerDescriptor = MTLSamplerDescriptor()
```

We can select between normalized and absolute coordinates with the `normalizedCoordinates` property. Most often, we want to use normalized coordinates, so this property is `true` by default.

```
samplerDescriptor.normalizedCoordinates = true
```

Next, we need to choose which filter mode to use when magnifying and minifying. Magnification happens when there are fewer texels than pixels in a region; minification happens when there are more texels than pixels. We saw the `MTLSamplerMinMagFilter` above; this is where we use it. The `.linear` enumerant enables bilinear interpolation.

```
samplerDescriptor.magFilter = .linear
samplerDescriptor.minFilter = .linear
```

We also select an address mode for each axis of texture space. Since we are working with 2D textures for now, we set the `sAddressMode` and `tAddressMode` to repeat:



```
samplerDescriptor.sAddressMode = .repeat  
samplerDescriptor.tAddressMode = .repeat
```

We can now make a sampler state by calling the

`makeSamplerState(descriptor:)` on a device:

```
samplerState = device.makeSamplerState(descriptor:  
samplerDescriptor)!
```

## ***Using Samplers and Textures in Shaders***

We will update our vertex structures once again to match the set of attributes we are using in our vertex descriptor:

```
struct VertexIn {  
    float3 position [[attribute(0)]];  
    float3 normal [[attribute(1)]];  
    float2 texCoords [[attribute(2)]];  
};  
  
struct VertexOut {  
    float4 position [[position]];  
    float3 normal;  
    float2 texCoords;  
};
```

To pass the texture coordinates through to our fragment function, we make a small update to our vertex function:

```
out.texCoords = in.texCoords;
```

The biggest updates are to the fragment function. We need to take a sampler object and a texture object so we can sample the texture at each fragment to determine its color.

Here is the updated fragment function signature:



```
fragment float4 fragment_main(
    VertexOut in [[stage_in]],
    texture2d<float, access::sample> textureMap [[texture(0)]],
    sampler textureSampler [[sampler(0)]])
```

Note the somewhat unusual syntax `texture2d<float, access::sample>`. The `texture2d` type is a template, which is akin to a Swift generic. Its template parameters indicate the type of component we want to receive (`float` or `half`) and how we will use it (`access::read`, `access::write` or `access::sample`). `texture(0)` indicates that we will bind the texture at fragment texture slot 0.

Buffers, textures, and samplers have separate binding index sets, so `sampler(0)` indicates that we will bind the sampler at fragment sampler slot 0. This doesn't conflict with texture slot 0 because of the difference in type.

To sample a texture, we use the `sample` method, passing our sampler and texture coordinates. We always get back a 4-element vector, regardless of how many components the source texture has.

```
float4 color = textureMap.sample(textureSampler, in.texCoords);
return color;
```

## ***Binding Samplers and Textures***

Like buffers, we bind samplers and textures on a render command encoder before drawing to use them in our shader functions. We haven't bound resources for use in a fragment shader before, but since we want to sample the texture for each fragment, we will do that now. To bind a sampler, we use the `setFragmentSamplerState(_:index:)` method, and to bind a texture, we use the `setFragmentTexture(_:index:)` method.

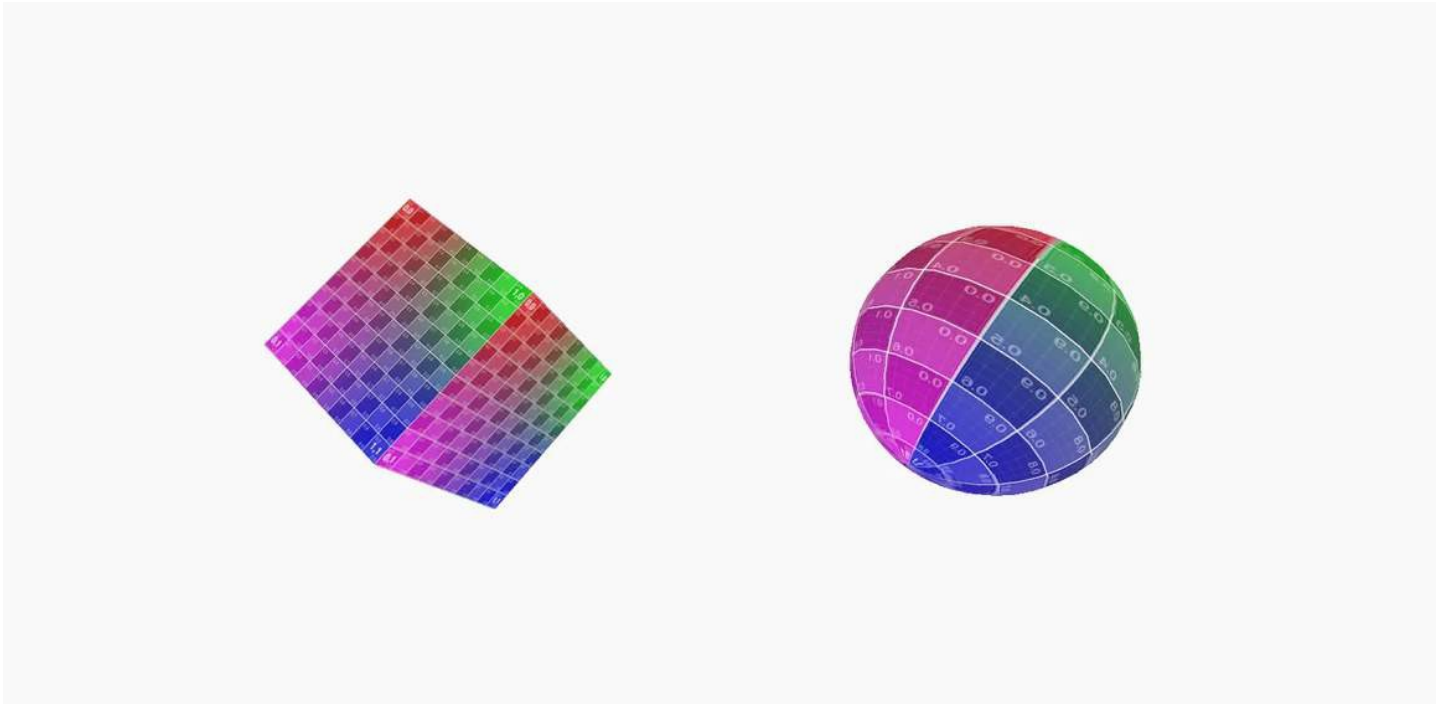
We bind each object at the slot we chose when updating our fragment function above:

```
renderCommandEncoder.setFragmentTexture(node.texture, index: 0)
```



```
renderCommandEncoder.setFragmentSamplerState(samplerState, index: 0)
```

We can update our sample app to draw a couple of different shapes to see how Model I/O parameterizes these shapes in texture space.



Soon, we will look at how to use Model I/O to load more complex 3D models, at which point we will unlock a new level of realism.

# Day 17: Assets



Warren Moore ·

5 min read · Apr 18, 2022

*This series of posts is my attempt to present the Metal graphics programming framework in small, bite-sized chunks for Swift app developers who haven't done GPU programming before.*

*If you want to work through this series in order, start [here](#). To download the sample code for this article, go [here](#).*

So far, when we have needed meshes to draw, we have asked Model I/O to generate basic 3D shapes. In practice, we often want to draw more sophisticated and detailed 3D models produced by an artist. In this article, we will look at how to start loading geometry and texture data from 3D model files.

An *asset* is any file that contains data authored separately from the application source code. This broad definition includes everything from images to sound clips to videos. A *3D model* is a type of asset that contains meshes, materials, animations, and other data that can be rendered in 3D. For the purposes of this article, we will use “3D model” and “asset” interchangeably.

There are many model file formats in current use: Wavefront OBJ, Khronos Group's glTF, Pixar's USD and Apple's USDZ, and Autodesk's FBX, to name a few. Each have different features and store data differently, but all are capable of storing node hierarchies, meshes, and materials. We will use OBJ in this article because of its simplicity. Be aware that in recent years, it has been superseded by more sophisticated formats like USD and glTF.

We will continue to use the `MDLMesh` and `MTKMesh` classes from Model I/O



and MetalKit to store mesh data, but we will get our meshes in a different way this time. We will use the `MDLAsset` class to load a 3D mesh and its associated texture, then convert them into forms that are suitable for rendering with Metal.

## ***Getting 3D Models***

Even if you can't afford an artist to build custom 3D models for you, there are many sources of free and affordable models online. The largest of these is [Sketchfab](#). Another, newer entrant is [Poly Haven](#). Consider browsing these sites and finding a model you like, to customize the sample app to your liking.

I have selected a free model named “Spot,” created by [Keenan Crane](#) of Carnegie Mellon University.



Spot the cow is a simple model consisting of a single mesh and a single texture, which makes her a perfect candidate for our initial exploration of asset loading with Model I/O.

## ***Introducing MDLAsset***

`MDLAsset` is a versatile class that can hold many different types of objects: lights, cameras, meshes, materials, and more.

To create an asset, we need three things: a file URL that points to the model

on disk, a vertex descriptor, and a buffer allocator.

As the mesh creation code is now somewhat lengthy, I extracted it into a method named `loadAsset()`.

We have used vertex descriptors and buffer allocators extensively, but I'll repeat the variable declarations here for completeness:

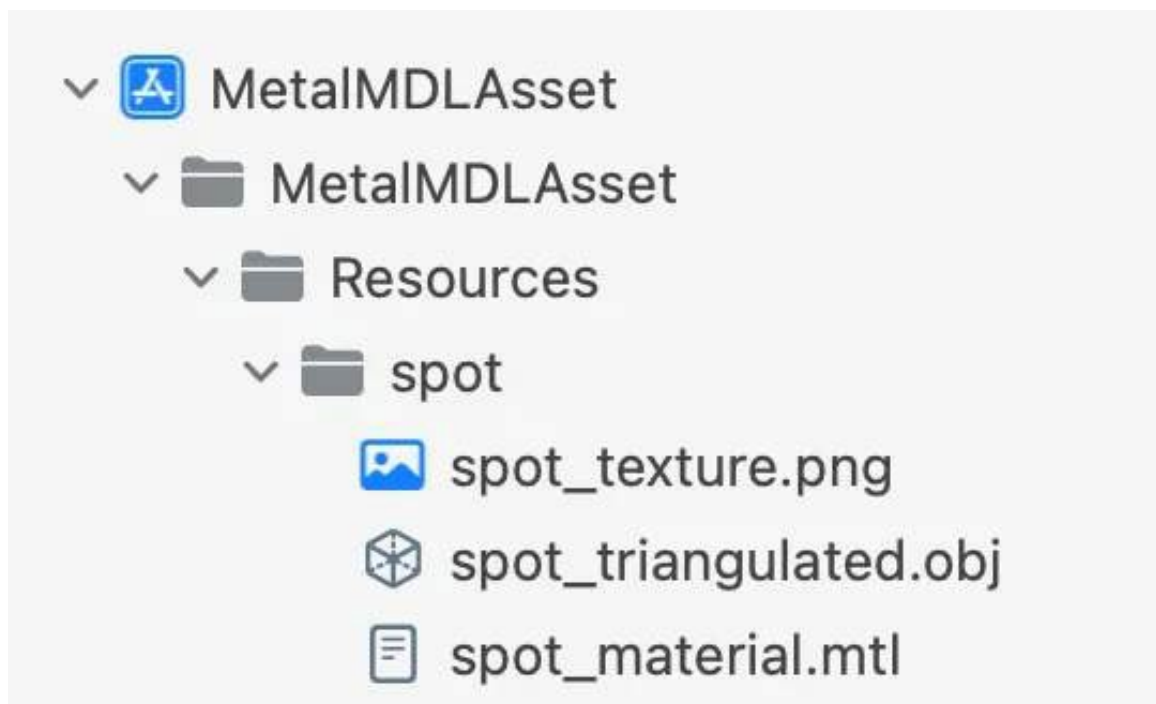
```
let allocator = MTKMeshBufferAllocator(device: device)

let mdlVertexDescriptor = MDLVertexDescriptor()
mdlVertexDescriptor.vertexAttributes[0].name =
MDLVertexAttributePosition
mdlVertexDescriptor.vertexAttributes[0].format = .float3
mdlVertexDescriptor.vertexAttributes[0].offset = 0
mdlVertexDescriptor.vertexAttributes[0].bufferIndex = 0
mdlVertexDescriptor.vertexAttributes[1].name =
MDLVertexAttributeNormal
mdlVertexDescriptor.vertexAttributes[1].format = .float3
mdlVertexDescriptor.vertexAttributes[1].offset = 12
mdlVertexDescriptor.vertexAttributes[1].bufferIndex = 0
mdlVertexDescriptor.vertexAttributes[2].name =
MDLVertexAttributeTextureCoordinate
mdlVertexDescriptor.vertexAttributes[2].format = .float2
mdlVertexDescriptor.vertexAttributes[2].offset = 24
mdlVertexDescriptor.vertexAttributes[2].bufferIndex = 0
mdlVertexDescriptor.bufferLayouts[0].stride = 32

vertexDescriptor =
MTKMetalVertexDescriptorFromModelIO(mdlVertexDescriptor)!
```

We create an `MTKMeshBufferAllocator` so Model I/O can write model data directly into `MTLBuffer`s. Then we define a vertex descriptor that has positions, normals, and texture coordinates, just as we did when generating our sphere and box meshes. Finally, we convert the Model I/O vertex descriptor to an `MTLVertexDescriptor` and store it to use when we create our render pipeline state.

Now we just need an asset URL. To make a model available to your app, add it to your project and make sure it is included in the app target's Copy Files phase. In the case of the Spot model, there are three files: an OBJ file containing the mesh, a PNG file containing the texture, and an MTL file containing material definitions.



Model I/O will automatically find other files referenced by a model, so we only need to create a URL pointing to the main OBJ file:

```
let assetURL =  
    Bundle.main.url( forResource:  
        "spot_triangulated", withExtension:  
        "obj")
```

Now we can instantiate an `MDLAsset` with these parameters:

```
let mdlAsset = MDLAsset(url: assetURL,  
                        vertexDescriptor: mdlVertexDescriptor,  
                        bufferAllocator: allocator)
```

`MDLAsset` does not automatically “resolve” texture file references, so we call `loadTextures()` to ask it to search through the materials in the file and locate any image files referenced by them.

```
mdlAsset.loadTextures()
```

Since a Model I/O asset can contain many different types, we need to ask

explicitly for objects by their type. We do this by calling the

`childObjects(of:)` method:

```
let meshes = mdlAsset.childObjects(of: MDLMesh.self) as? [MDLMesh]
guard let mdlMesh = meshes?.first else {
    fatalError("Did not find any meshes in the Model I/O asset")
}
```

To load the textures referenced by the model file, we will again use

`MTKTextureLoader`.

```
let textureLoader = MTKTextureLoader(device: device)
let options: [MTKTextureLoader.Option : Any] = [
    .textureUsage : MTLTextureUsage.shaderRead.rawValue,
    .textureStorageMode : MTLStorageMode.private.rawValue,
    .origin : MTKTextureLoader.Origin.bottomLeft.rawValue
]
```

Materials in Model I/O are represented by the `MDLMaterial`. Each material has numerous instances of `MDLMaterialProperty`, each one representing a material property such as base color, roughness, or metalness.

For the moment, we will assume that each mesh has a single material, held by its first submesh.

```
let firstSubmesh = mdlMesh.submeshes?.firstObject as? MDLSubmesh
let material = firstSubmesh?.material
```

Specifically, we care about its base color. If this material property contains a texture, we get a file URL pointing to it and use our texture loader to turn it into an `MTLTexture`:

```

var texture: MTLTexture?
if let baseColorProperty =
    material?.property( with:
        MDLMaterialSemantic.baseColor)
{
    if baseColorProperty.type == .texture,
        let textureURL = baseColorProperty.urlValue
    {
        texture = try?
            textureLoader.newTexture( URL:
                textureURL,
                options: options)
    }
}

```

Assuming our mesh and texture were loaded successfully, we can convert the mesh to an `MTKMesh`, which we already know how to draw.

```

let mesh = try! MTKMesh(mesh: mdlMesh, device: device)

```

We create a node to hold the mesh, and assign the base color texture to it. Then we hold references to it so we can update and render it later in our draw method.

```

cowNode = Node(mesh: mesh)
cowNode.texture = texture

nodes = [cowNode]

```

This concludes our `loadAsset()` method. The rest of the code is largely unchanged.

Running the sample app, we can see Spot the cow spinning in all her textured glory.

We have now used textures to add a bit more surface detail and realism to our scenes, but we are still far from photorealism. In the next article, we





will consider how to add lights to our scenes.

# Day 18: Directional Light



Warren Moore ·

16 min read · Apr 27, 2022

*This series of posts is my attempt to present the Metal graphics programming framework in small, bite-sized chunks for Swift app developers who haven't done GPU programming before.*

*If you want to work through this series in order, start [here](#). To download the sample code for this article, go [here](#).*

Last time, we enhanced the visual complexity of our 3D meshes by applying textures to them. In this article, we will start to look at how we handle light and its interaction with surfaces.

When we look at the world, we do not see objects themselves, but rather the light emitted, reflected, and transmitted by objects.

Most of the parcels of light — photons — that arrive at our eyes were not emitted directly by a light source. Instead, photons bounce off of objects, get absorbed and then reemitted by objects, and get refracted through transparent objects. Most of the light that we see has taken a complex path through the world to get to us, whether it originated from a distant star or a fluorescent lamp in our living room.

When we draw 3D scenes, we are responsible for creating lights to illuminate our virtual objects. Our model of light is necessarily simpler than light in the real world. In fact, for the purposes of this article, we will only consider two types of lights: directional and ambient.

*A directional light* is a light that is so distant that its rays are all aligned along a single direction. As an example, consider the sun. Because we are so far

away from the sun, the direction of sun rays at a particular point on Earth all seem to be parallel; this is what produces sharp, distinct shadows on a sunny day. Since all of the light arrives along a single direction, we don't need to consider the position of a directional light, just its direction and intensity.

By contrast, a *point light* is a light with a definite position that casts rays in all directions. For example, we might model a light bulb as a point light. We will not discuss point lights further in this article.

For much of the history of graphics, calculating how light bounces around the world was too expensive to perform in real-time. This led to a huge number of approximation techniques classified as *global illumination* or *indirect illumination*. These techniques range widely in complexity and computational cost, but all of them account for light that has bounced multiple times before arriving at the virtual camera.

In this article, we will consider a very simple form of global illumination: ambient light. An *ambient light* is one that has neither position nor direction, only intensity. Ambient light is meant to account for all of the light in a scene that arrives at the camera after having bounced more than once. Reducing all of the bounced light to a single constant intensity is highly non-realistic, but in the absence of a better light model, it helps ease the harshness of pure direct lighting.

## A Light Class

Our first swing at designing a light class will be very simple. We have an enumeration, `LightType`, that includes our two light types:

```
enum LightType : UInt32
{
    case ambient
    case directional
}
```

Our `Light` class has a `type` member defining its type, an RGB color, a scalar intensity, and a direction.

```
class Light {
    var type = LightType.directional
    var color = SIMD3<Float>(1, 1, 1)
    var intensity: Float = 1.0
    var direction = SIMD3<Float>(0, 0, -1)
}
```

We separate the color from the intensity because it is often useful to select the *tone* of a light separate from its *brightness*. These values are multiplied together when passed into the shader. At this time, we will not consider the units of light intensity; we will empirically choose values that make our scene look good.

Since we are not yet modeling point lights, we do not have a member representing position. Instead, the `direction` member represents the direction of a directional light. Direction is ignored for ambient lights.

We can store a list of lights in our scene by adding a member to our renderer class:

```
var lights = [Light]()
```

## ***Introducing Multiple Constant Data Types***

We have arrived at the point where we need to take multiple kinds of constant data as parameters to our shader functions. We will have three different kinds of constant data: light constants, frame constants, and node constants.

We separate the data in this way because node data changes every draw call, while other data — such as the projection matrix and the light list — remain the same across all draw calls in a frame.

Each kind of data has its own struct in both Swift and MSL. The Swift definitions look like this:

```

struct LightConstants {
    var intensity: simd_float3
    var direction: simd_float3
    var type: UInt32
}

struct NodeConstants {
    var modelViewMatrix: float4x4
}

struct FrameConstants {
    var projectionMatrix: float4x4
    var lightCount: UInt32
}

```

The Metal shading language definitions look like this:

```

struct Light
{
    float3
    intensity;
    float3 direction;
    LightType type;
};

struct NodeConstants
{
    float4x4
    modelViewMatrix;
};

struct FrameConstants
{
    float4x4
    projectionMatrix; uint

```

These structures are designed carefully to minimize discrepancies in padding and alignment between Swift and MSL. These languages do not have the same type layout rules, so we must continually be conscious of how things are laid out on both sides of the language gap. (See more on Swift’s layout rules [here](#) and some comments by Jordan Rose [here](#)).

## ***A More General Approach to Constant Buffer***

***Allocation*** Even though we now have multiple types of constant data changing at different frequencies, we still only need to allocate one constant buffer.

Rather than basing the size of the buffer on the number of objects in the scene, we will simply choose a size that is large enough to hold all the constant data we could possibly need over a span of three frames. In the eventuality where we

want to draw more objects, we simply make the size

bigger and recompile.

We eliminate the `constantSize` and `constantStride` members in favor of a global variable that stores the constant buffer's entire size:

```
let MaxConstantsSize = 1_024 * 1_024
```

Our `makeResources()` method simplifies to:

```
func makeResources() {  
    constantBuffer = device.makeBuffer(length: MaxConstantsSize,  
                                       options: .storageModeShared)  
}
```

We now need a scheme for allocating regions of this buffer for constant storage. This scheme must meet two requirements:

1. The buffer is a ring buffer: when we reach the end of the buffer, we must wrap back around to the beginning. In order to avoid data corruption, the size of the buffer must be at least three times larger than our maximum per-frame required space.
2. We must honor the alignment requirements of the types we write into the buffer, as well as the alignment requirements of the hardware itself. Therefore, when allocating, we must choose the smallest alignment that satisfies both of these (potentially different) alignments.

Fortunately, these requirements are not terribly hard to satisfy, especially if we adequately anticipate the required size of the buffer.

We continue to store the byte offset at which the next constant data will be written in the `currentConstantBufferOffset` member. We write a new method, `allocateConstantStorage(size:alignment:)` that does the necessary math to reserve a portion of the constant buffer for updating.





```

func allocateConstantStorage(size: Int, alignment: Int) -> Int
{
    let effectiveAlignment = lcm(alignment, MinBufferAlignment)
    var allocationOffset = align(currentConstantBufferOffset, upTo:
effectiveAlignment)
    if (allocationOffset + size >= MaxConstantsSize)
        { allocationOffset = 0
    }
    currentConstantBufferOffset = allocationOffset + size
    return allocationOffset
}

```

Briefly, this method finds an alignment that accommodates type and hardware requirements, aligns the offset up to this alignment, verifies that we won't overflow the buffer by writing into it at this point, and then bumps the current offset up to make room for the newly allocated region. If we would overflow the buffer, we instead wrap the offset back to the beginning of the buffer, thus implementing the circular buffer requirement.

There is one kind of buffer overrun that we haven't checked for. If the size to be allocated exceeds the length of the entire buffer, we cannot possibly accommodate such an allocation. Since one of the assumptions we made is that the buffer has been allocated with adequate space for three frames' worth of data, this situation is assumed never to arise.

## ***Updating Constant Data***

Now, at the beginning of our draw method, we can update our various kinds of constants:

```

func draw(in view: MTKView)
{
    frameSemaphore.wait()

    updateLightConstants()
    updateFrameConstants()
    updateNodeConstants()
    //...
}

```

Since we want to accommodate situations where the number of lights changes from frame to frame, we place the lights in their own separate region of the constant buffer. To make sure we have enough room to store

all of the lights, we request a region of the buffer that is the stride of `LightConstants` multiplied by the current number of lights. We then iterate over the lights and write them each into the buffer. We store the base offset in a new member named `lightConstantsOffset` so we can retrieve it later when drawing.

```
func updateLightConstants() {
    let layout = MemoryLayout<LightConstants>.self

    lightConstantsOffset = allocateConstantStorage(
        size: layout.stride * lights.count,
        alignment: layout.stride)

    let lightsBufferPointer = constantBuffer.contents()
        .advanced(by: lightConstantsOffset)
        .assumingMemoryBound(to: LightConstants.self)

    for (lightIndex, light) in lights.enumerated()
    { lightsBufferPointer[lightIndex] =
        LightConstants(intensity: light.color * light.intensity,
            direction: light.direction,
            type: light.type.rawValue)
    }
}
```

Our frame constants are the projection matrix and the light count. We can compute these on the fly, then use our new allocation method to write them into the constant buffer.

```
func updateFrameConstants() {
    time += (1.0 / Double(view.preferredFramesPerSecond))

    let aspectRatio =
        Float(view.drawableSize.width / view.drawableSize.height)

    let projectionMatrix =
        simd_float4x4( perspectiveProjectionFoVY: .
            pi / 3, aspectRatio: aspectRatio,
            near: 0.01,
            far: 100)

    var constants =
        FrameConstants( projectionMatrix:
            projectionMatrix, lightCount:
            UInt32(lights.count))

    let layout = MemoryLayout<FrameConstants>.self

    frameConstantsOffset = allocateConstantStorage(
        size: layout.size, alignment: layout.stride)

    let constantsPointer = constantBuffer.contents()
        .advanced(by: frameConstantsOffset)

    constantsPointer.copyMemory(from: &constants,
```



```
        byteCount: layout.size)
    }
```

Finally, we update our node constants. We split the projection matrix off from the other transforms, so the transform matrix we write out for each node is its model-view matrix, the product of the view matrix and the node's model-to-world matrix.

In the `updateNodeConstants()` method, we calculate our camera matrix as usual, and also perform any per-node animations we might want. In this sample, we will spin Spot the cow around her vertical axis so we can see the lighting update as she moves.

```
func updateNodeConstants()
{
    let t = Float(time)

    let cameraPosition = SIMD3<Float>(0, 0.5, 2)
    let viewMatrix = simd_float4x4(translate: -cameraPosition)

    let rotationAxis = normalize(SIMD3<Float>(0, 1, 0))
    let rotation = simd_float4x4(rotateAbout: rotationAxis,
                                byAngle: t)

    cowNode.transform = rotation

    nodeConstantsOffsets.removeAll()
    for node in nodes {
        let transform = viewMatrix * node.worldTransform
        var constants = NodeConstants(modelViewMatrix: transform)

        let layout = MemoryLayout<NodeConstants>.self
        let offset = allocateConstantStorage(
            size: layout.size,
            alignment: layout.stride)

        let constantsPointer = constantBuffer.contents()
            .advanced(by: offset)

        constantsPointer.copyMemory(from: &constants,
                                    byteCount: layout.size)

        nodeConstantsOffsets.append(offset)
    }
}
```

As with the lights and frame constants, we store each node's buffer offset so we can use it later when drawing.

Now that we have a way to store lights and get their data into our constant

buffer, let's talk about our actual lighting model.

## ***Ambient Light***

The model we use for calculating ambient lighting is extremely simple. Since the ambient light intensity is directionless and constant throughout the scene, we determine its contribution to the final reflected color by just multiplying it by the surface's reflectance.

The following pseudocode shows how to accumulate the contribution of an ambient light.

```
litColor += ambientIntensity * baseColor
```

## ***Specular and Diffuse Reflectance***

Once we move beyond the simple model of ambient light, we must consider how light actually bounces around the world. We can divide reflection into two types: *specular reflection* and *diffuse reflection*. We will also split materials into two categories: *metals* and *dielectrics*.

Specular reflection makes surfaces appear shiny by producing highlights and mirror-like reflected images. A smooth metallic surface will produce an almost perfect reflection, while a less smooth surface, like the dull side of a sheet of aluminum foil, produces much less distinct reflections.

Metals and dielectrics both exhibit specular reflection. Shiny plastic, polished leather, and glazed pottery can all have distinct specular highlights. Even so, most of the light reflected by dielectrics is reflected diffusely. Metals, by contrast, *only* reflect specularly, not diffusely.

Diffuse reflection results when light beams bounce off a surface in seemingly random directions that are not highly correlated in a particular direction. Canvas and cardboard are two materials whose appearance is dominated by diffuse reflection. When we think of a non-metallic object's color, we are mostly thinking of the way it diffusely reflects light.

# Theory of Diffuse Reflectance

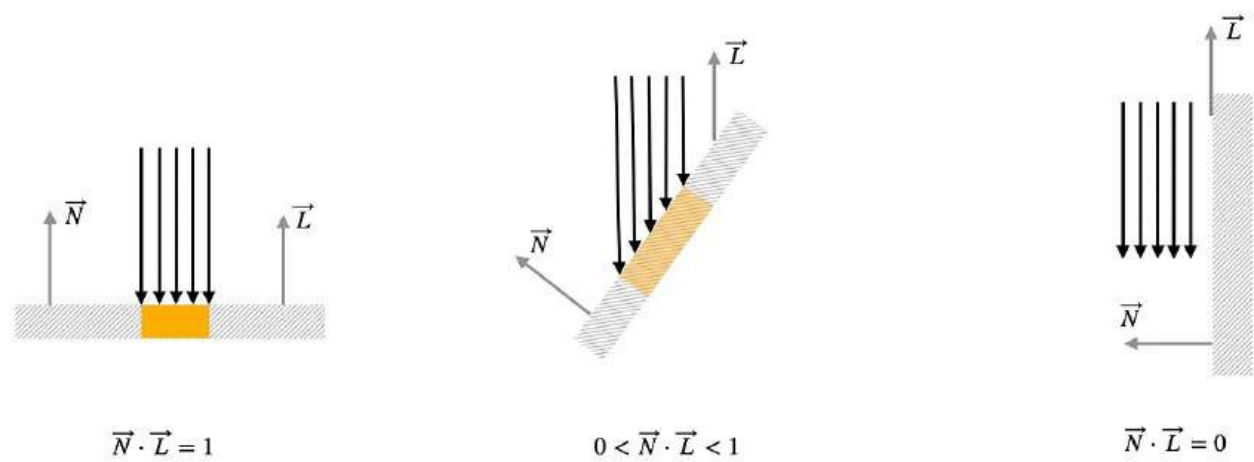
Now that we know about the different types of reflection, let's look deeper into the physical phenomena of reflection so we can figure out how to calculate it ourselves.

Diffuse reflection occurs when light rays enter a dielectric material, bounce around, and then exit some short time later. Having bounced around, light exits the surface in a somewhat random direction. What effect does this randomization have on the light we see?

The answer is that light appears to be reflected uniformly from a given surface point. In other words, the direction of a light ray leaving a surface is not correlated with the original light ray that arrived at the surface. This also means that the amount of light diffusely reflected at a given point is not dependent on where the camera is positioned.

The amount of light reflected diffusely *does* depend on the alignment between the surface normal and the light direction, though. In particular, the more aligned the normal and light direction, the more light will be reflected, simply because more light is arriving at each nearby point.

To illustrate this, consider a number of parallel light beams arriving at a surface. In the figure below, the surface normal is denoted as  $\vec{N}$  and the light direction (the direction *toward* the light) is denoted as  $\vec{L}$ .



In the left part of the figure, the light beams are arriving exactly parallel to

the surface normal. In the middle of the figure, the surface has tilted away somewhat from the light direction. In this arrangement, even though the same amount of light arrives on the surface, it is spread over a wider area, meaning the surface will appear proportionately dimmer. In the right of the figure, the surface is oriented perpendicular to the light direction, and no light hits the surface at all, meaning it will be completely dark (remember that we're ignoring indirect, bounced light).

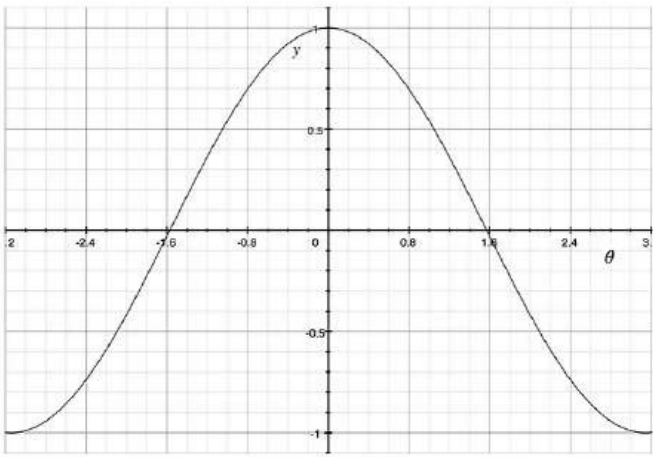
How can we capture this phenomenon mathematically? It turns out that Lambert's cosine law, formulated in the mid-18th century, provides an answer: the luminous intensity reflected diffusely from a surface is proportional to the cosine of the angle between the direction of the light and the surface normal. From vector algebra, we know that the cosine of the angle between two normalized vectors is exactly their dot product.

Assuming we have a normalized surface normal and a normalized light direction, we can implement diffuse reflection for a directional light with the following pseudocode:

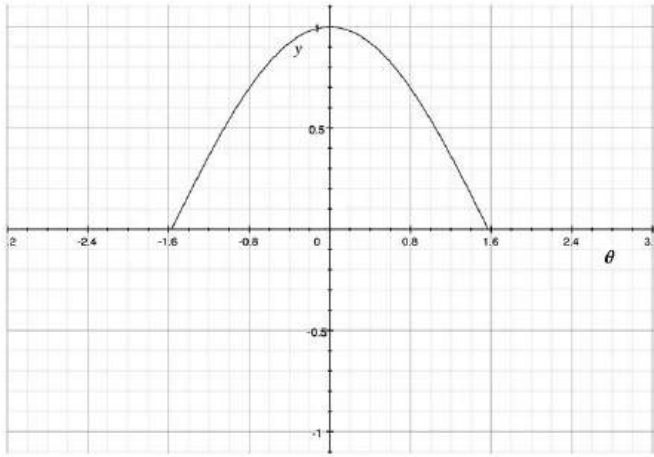
```
diffuseFactor = max(0, dot(N, L))
litColor += diffuseFactor * lightIntensity * baseColor
```

The `dot` function produces a value between -1 and 1 when operating on normalized vectors. Values less than 0 occur when the light is *behind* the surface and should not contribute to the surface's illumination. We “clamp” values less than 0 to 0 using the `max` function.

The figure below shows the unclamped and clamped cosine function.



$$y = \vec{N} \cdot \vec{L} = \cos \theta$$



$$y = \vec{N} \odot \vec{L} = \max(0, \cos \theta)$$

We’re playing pretty fast and loose with the math here: there’s actually a missing factor of  $1/\pi$  that is absent from the code above. This will become important when we start to do physically-based rendering, but until that time, we’ll fake it like they did in the bad old days.

## Theory of Specular Reflectance

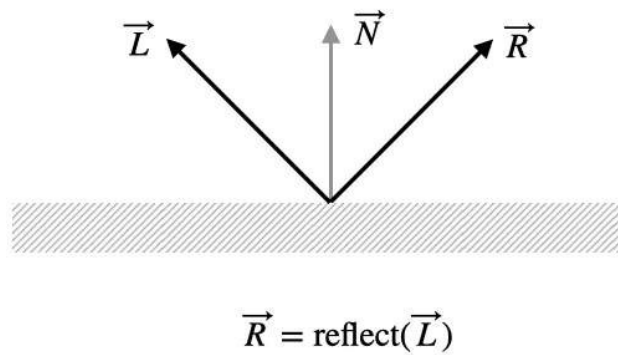
In contrast to diffuse reflectance, specular reflectance produces highlights, because light is reflected mostly along one dominant direction. But what is this direction, and how do we model highlights mathematically and in code?

Our answer comes from the *Blinn specular model*. Formulated by Jim Blinn in 1977, the model improved on prior work done by Phong.

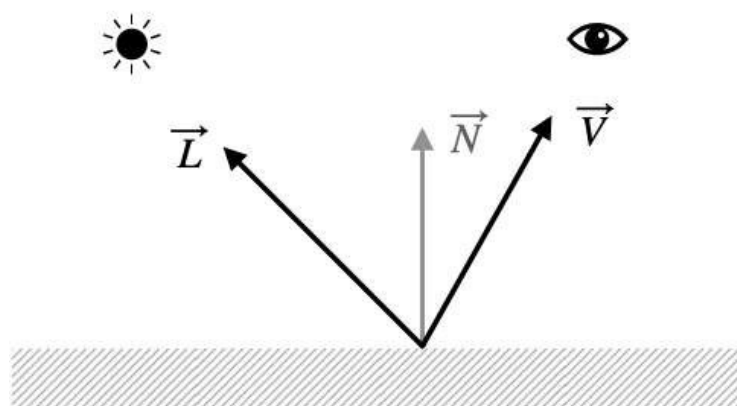
First, we define the ideal mirror reflection vector  $\mathbf{R}$  by reflecting the light direction around the surface normal. Imagine shining a flashlight into a mirror. If you situate yourself so that the flashlight appears at maximum brightness, the angle your line of sight makes with the mirror will be equal to the angle between the flashlight’s beam and the mirror.

We assume the existence of a function called `reflect` that calculates this vector for us. The relationship between the  $\mathbf{L}$  vector and  $\mathbf{R}$  vector is shown in the diagram below.

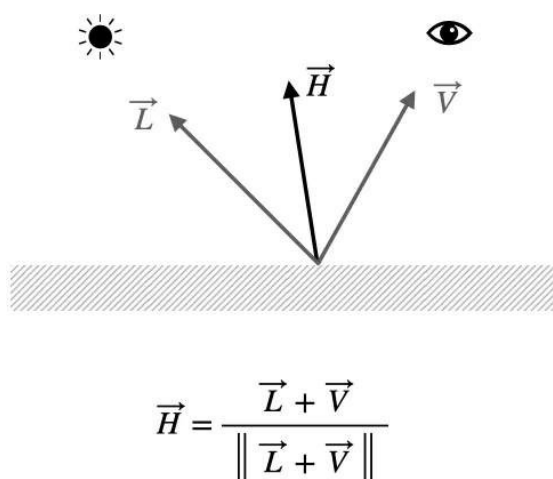




We also need to define a vector  $\vec{v}$  that points along the virtual camera’s line of sight. Since the appearance of specular highlights depends on our point of view, this vector will factor into our lighting calculations.



The Blinn specular model calls for a *halfway vector* that is the average of the light direction  $\vec{L}$  and the view vector  $\vec{v}$ .



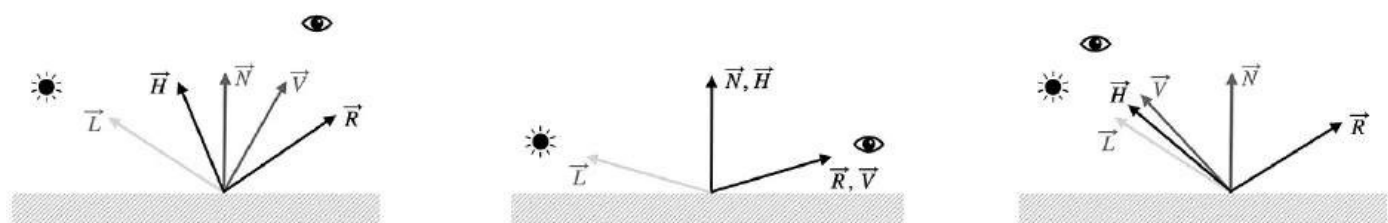
Thinking back to our flashlight and mirror example, we observed that specular highlights attain maximum brightness when  $\vec{v}$  is aligned with  $\vec{R}$ . Blinn noticed that the dot product between  $\vec{v}$  and  $\vec{R}$  is closely correlated with the dot product

between the surface normal  $\mathbf{N}$  and the halfway vector

H.

Several lighting and viewing situations are illustrated below. Note that in each case, the angle between  $\vec{V}$  and  $\vec{R}$  is similar to the angle between  $\vec{N}$  and

$\vec{H}$ :

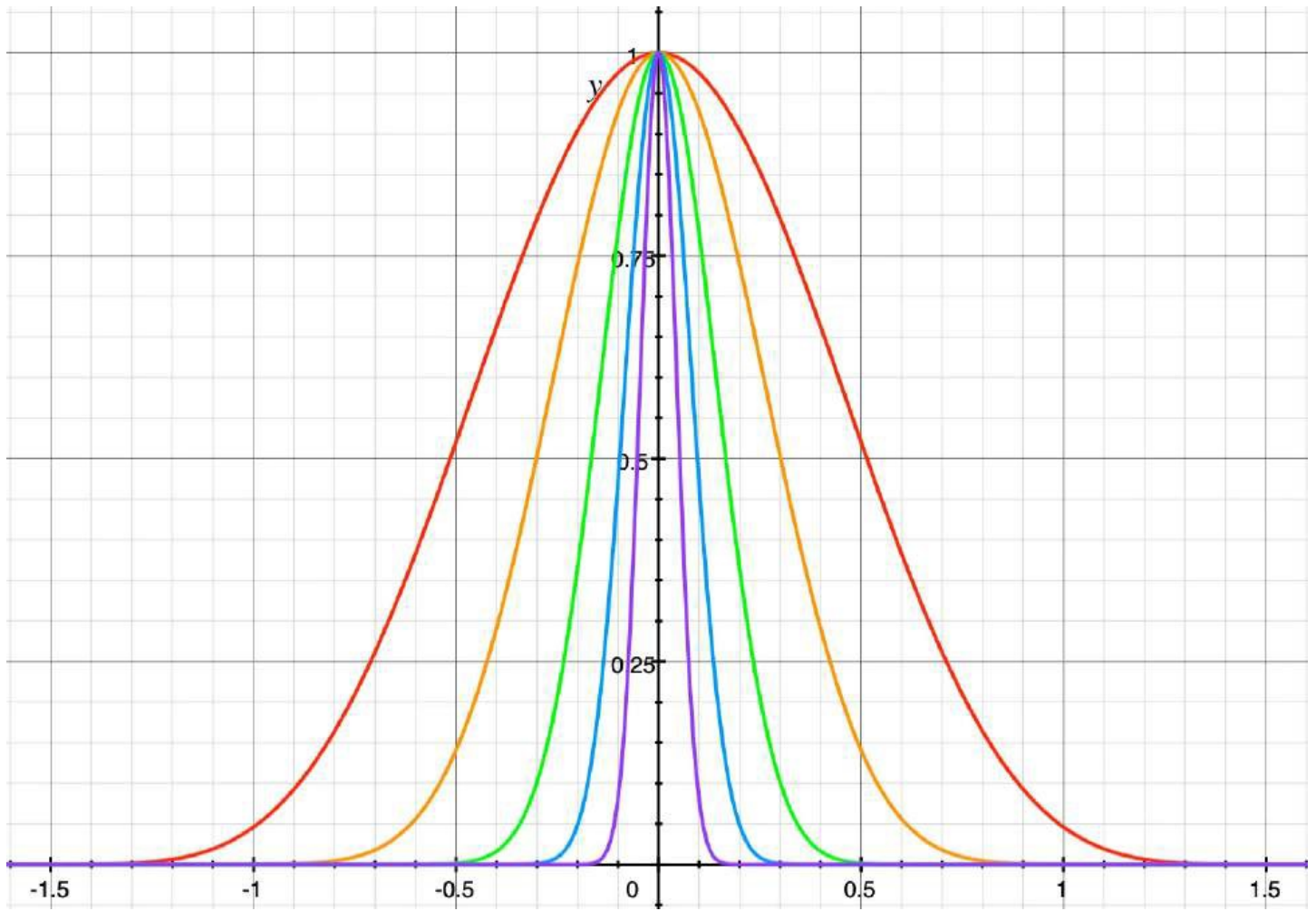


$$\vec{N} \cdot \vec{H} \approx \vec{V} \cdot \vec{R}$$

Simply taking the dot product between  $\vec{N}$  and  $\vec{H}$  gives us an approximation of how closely aligned the view direction is with the specular reflection, but we need another parameter to control how “tight” the highlight is. This allows us to simulate different degrees of surface roughness: a shiny surface has tighter and brighter highlights than a rough surface.

Following the work of Phong, Blinn models this effect with a *specular exponent*. We raise the dot product of  $\vec{N}$  and  $\vec{H}$  to the power of the specular exponent  $s$ , which has the effect of narrowing the highlight as the exponent increases.

The figure below shows a clamped cosine function raised to several different exponents (5, 15, 50, 150, and 500). As the exponent increases, the function gets steeper and draws toward the vertical axis.



The following pseudocode shows how to implement Blinn specular lighting.

```
H = normalize(L + V)
specularFactor = powr(saturate(dot(N, H)), specularExponent)
```

The `saturate` function clamps its parameter between 0 and 1. The `powr` function raises its first parameter to the power of its second parameter.

Now that we have the various bits of math we need to implement lighting, let's update our shaders to use our light data to illuminate our scene.

## Updating the Vertex Shader

We will perform our lighting in *view space*. This means that all of the vectors we work with need to be transformed into the coordinate system of the camera.

To aid this, we add a `viewPosition` member to our output vertex structure. This will store the vertex's position in view space.



```
struct VertexOut {
    float4 position [[position]];
    float3 viewPosition;
    float3 normal;
    float2 texCoords;
};
```

In the vertex function, we transform the normal by the model-view matrix before passing it along. We also calculate the view-space position of the vertex as an intermediate vector so we can pass it to the fragment function and also transform it into clip space with the projection matrix.

Here is the complete modified vertex function.

```
vertex VertexOut vertex_main(
    VertexIn in [[stage_in]],
    constant NodeConstants &node [[buffer(2)]],
    constant FrameConstants &frame [[buffer(3)]]
)
{
    float4 viewPosition = node.modelViewMatrix *
                          float4(in.position, 1.0);
    float4 viewNormal = node.modelViewMatrix *
                       float4(in.normal, 0.0);

    VertexOut out;
    out.position = frame.projectionMatrix * viewPosition;
    out.viewPosition = viewPosition.xyz;
    out.normal = viewNormal.xyz;
    out.texCoords = in.texCoords;
    return out;
}
```

## ***Implementing a Lighting Fragment Function***

To mirror our changes to the constant buffer, we add new parameters to the fragment function that take the frame constants and the light list:

```
fragment float4 fragment_main(
    VertexOut in [[stage_in]],
    constant NodeConstants &node [[buffer(2)]],
    constant FrameConstants &frame [[buffer(3)]],
    constant Light *lights [[buffer(4)]],
    texture2d<float, access::sample> textureMap [[texture(0)]],
    sampler textureSampler [[sampler(0)]]
)
```

In the fragment function, we first sample the texture to determine the surface's base color:

```
float4 baseColor = textureMap.sample(textureSampler, in.texCoords);
```

For the time being, we will assume a constant specular exponent of 50. This could also be passed in as a node constant.

```
float specularExponent = 50.0;
```

To perform our lighting calculations, we need the **N**, **V**, **L** and **H** vectors discussed above. The **N** and **V** vectors are independent of light direction, so we compute them once outside the light loop.

```
float3 N = normalize(in.normal);  
float3 V = normalize(float3(0) - in.viewPosition);
```

Note that since the camera is at the origin in view space (by definition), we find **V** by taking the vector from the fragment's view-space position and the origin.

Now we create a variable called **litColor** that we will accumulate all of the light from the various lights in the scene.

```
float3 litColor { 0 };
```

To accumulate the lighting, we loop over each light in turn. Each light can have an ambient, diffuse, and specular contribution, so we set up a few variables to store these different terms, then index into the light list to get the light for the current iteration:

```

for (uint i = 0; i < frame.lightCount; ++i) {
    float ambientFactor = 0;
    float diffuseFactor = 0;
    float specularFactor = 0;

    constant Light &light = lights[i];

```

For ambient lights, the ambient factor is 1 and the diffuse and specular factors are 0:

```

switch(light.type) {
    case LightTypeAmbient:
        ambientFactor = 1;
        break;

```

For directional lights, we compute the diffuse factor and specular factor according to the Lambertian and Blinn models discussed in previous sections:

```

case LightTypeDirectional: {
    float3 L = normalize(-light.direction);
    float3 H = normalize(L + V);
    diffuseFactor = saturate(dot(N, L));
    specularFactor = powr(saturate(dot(N, H)),
specularExponent);
    break;
}
}

```

To end the light loop, we sum up the ambient, diffuse, and specular factors, multiply them by the current light's intensity, multiply by the base color, and add the result to our lit color:

```

litColor += (ambientFactor + diffuseFactor + specularFactor) *
light.intensity * baseColor.rgb;
}

```

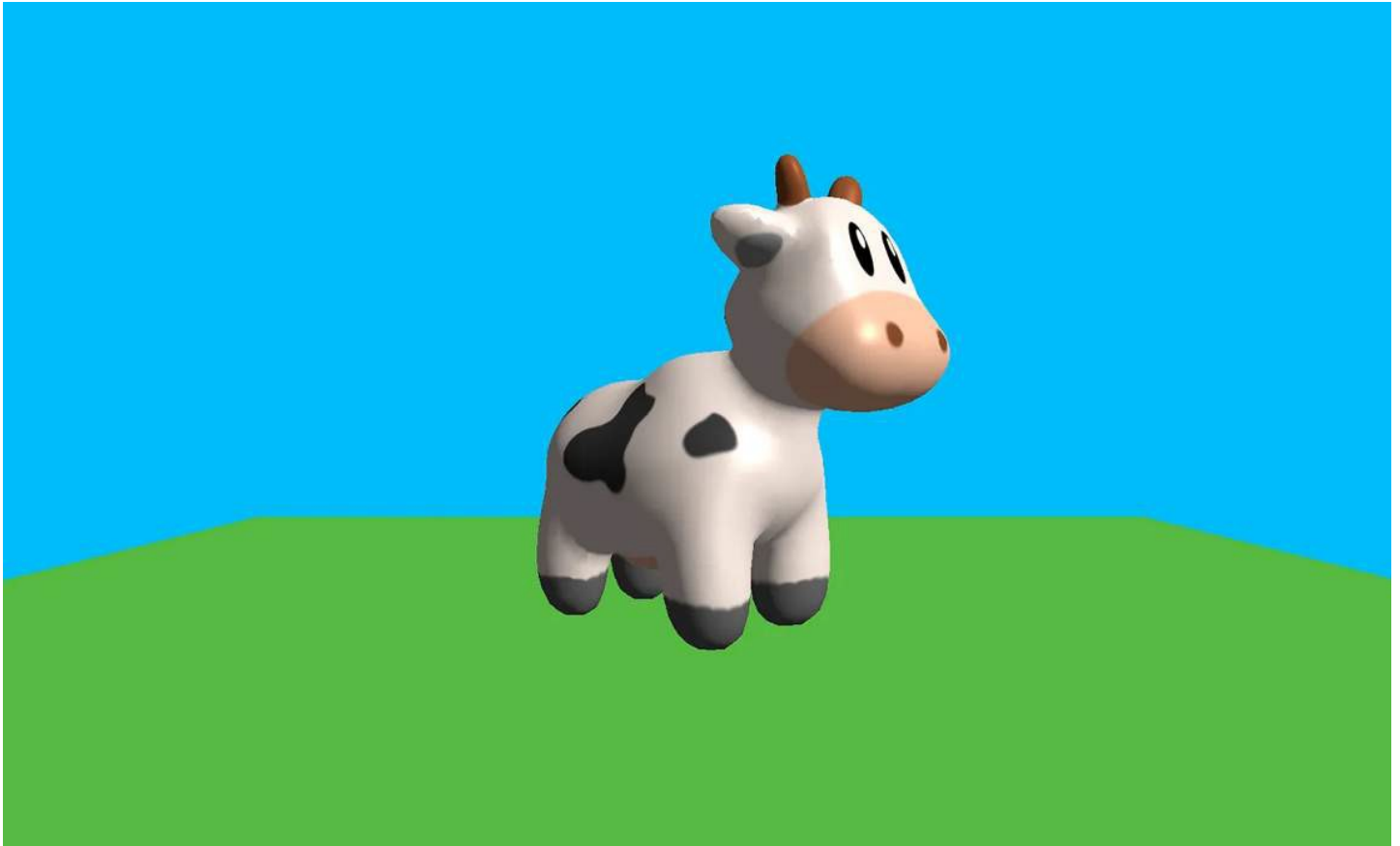
Finally, we return the lit color from the fragment function:





```
return float4(litColor * baseColor.a, baseColor.a);  
}
```

Running our sample app, we can see Spot the cow illuminated by an ambient light and a directional light:



Lighting is an important part of realism in any 3D scene, but of course, it is only part of the picture. Next time, we'll look at how to implement shadow mapping to make our 3D figures look more grounded.

# Day 19: Directional Shadows



Warren Moore ·

9 min read · Apr 28, 2022

*This series of posts is my attempt to present the Metal graphics programming framework in small, bite-sized chunks for Swift app developers who haven't done GPU programming before.*

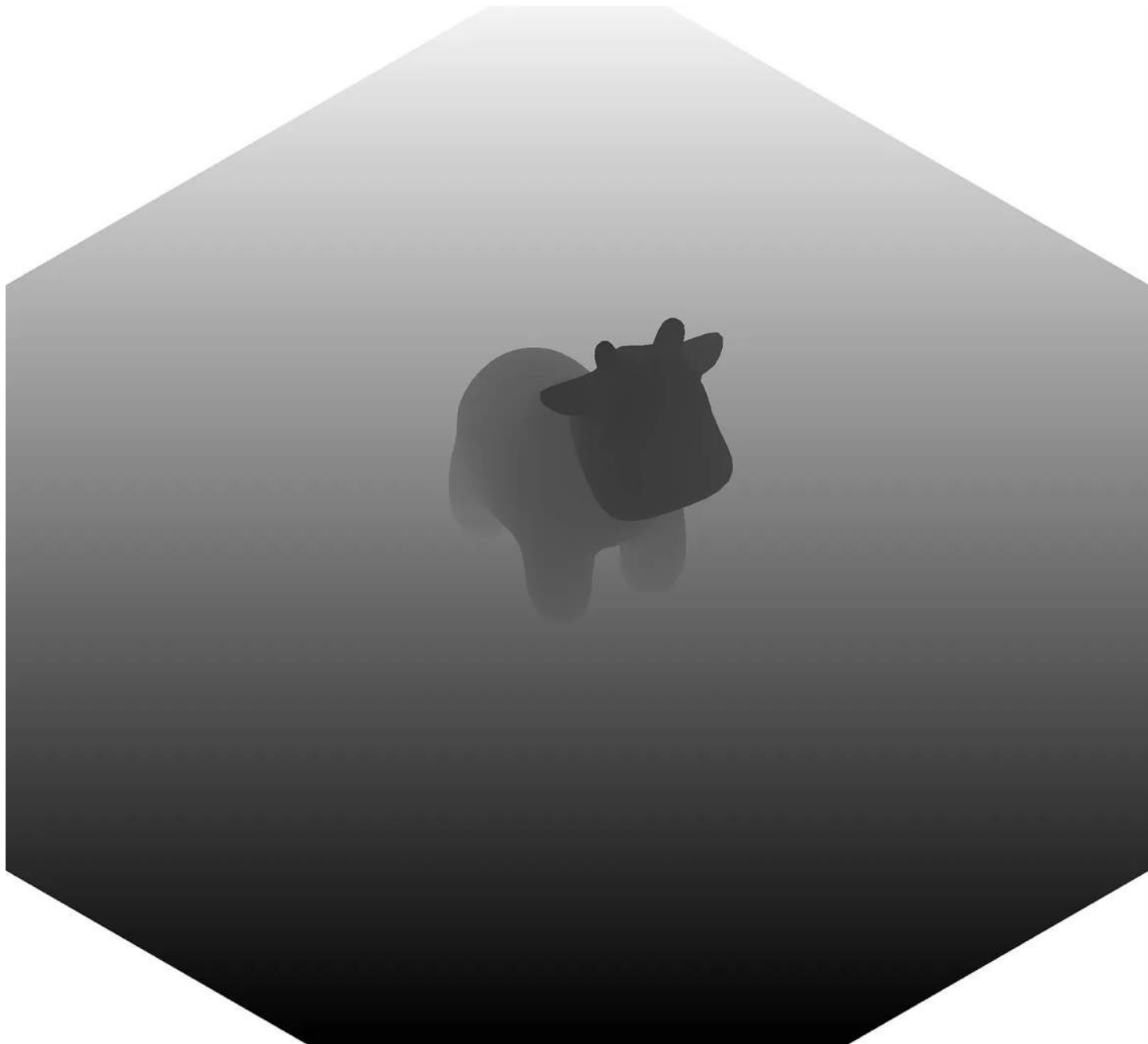
*If you want to work through this series in order, start [here](#). To download the sample code for this article, go [here](#).*

Adding lights to our scene certainly adds an element of realism, but there's a problem. Unlike in the real world, we don't get shadows for free. Instead, each triangle is shaded independently without regard for whether there is other geometry between it and the light.

To emulate shadows, we need some way to answer the question, "Does this point actually receive any light?" If the answer is no, the point should be in shadow. But how do we answer this question cheaply and accurately?

Our answer is shadow mapping. *Shadow mapping* is the procedure of rendering a depth map of the scene from the perspective of a light, then querying this depth map when drawing the scene to determine if a point is illuminated or in shadow. We call this kind of depth map a *shadow map*.

Here is a visualization of the shadow map we will learn to produce below.



This sounds simple enough, but there are some subtleties in the implementation we will walk through in this article.

## ***Positioning and Orienting Lights***

When we were rendering with directional lights without accounting for shadows, we didn't have a notion of light position, since all of the light's rays were assumed to be parallel. However, if we want to render a depth map from the light's perspective, we need to choose a point of view for it.

To extend our light abstraction for shadows and eventually point lights, we will replace the mutable `direction` member with a full model-to-world transform matrix. From this matrix, we can extract the location and direction of the light.

```
class Light {
    // ...
    var worldTransform: simd_float4x4 = matrix_identity_float4x4

    var position: SIMD3<Float> {
        return worldTransform.columns.3.xyz
    }

    var direction: SIMD3<Float> {
        return -worldTransform.columns.2.xyz
    }
    // ...
}
```

The position of the light is the translational component of its transform, while its direction is the negative-Z axis of its orientation.

## Look-At Transforms

Back on day 10, we learned about elementary transformations like translation, rotation, and scaling. We have since extended these operations with basic 3D transformations.

It turns out that there is a slightly more complex transformation type that is exceedingly useful: the look-at transformation.

To construct a look-at matrix, we need three parameters: the target point (what we’re looking *at*), the from point (where we’re looking *from*), and the “up” vector (which way is up). From these intuitive parameters, we can build a matrix that positions a camera or light and orients it toward a point of interest.

Mathematically, we follow a simplified form of Gram-Schmidt orthonormalization, first establishing the (negative) Z axis, then finding the X axis, and finally calculating the Y axis. We exploit the fact that the 3D vector cross product of two linearly-independent vectors is a third vector that is perpendicular to both. In code, it looks like this:

```
extension simd_float4x4
{ init(lookAt at:
    SIMD3<Float>,
        from: SIMD3<Float>,
        up: SIMD3<Float>)
{
```



```

        let x = normalize(cross(zNeg, up))
        let y = normalize(cross(x, zNeg))
        self.init(
            SIMD4<Float>(x, 0),
            SIMD4<Float>(y, 0),
            SIMD4<Float>(-zNeg, 0),
            SIMD4<Float>(from, 1)
        )
    }
}

```

With this look-at utility initializer, we can now more easily position our cameras and lights, without having to tediously combine matrices.

## ***Shadow Projection Matrices***

Just as we need a view matrix and projection matrix when rendering from the point of view of a camera, we need a view matrix and projection matrix for a shadow-casting light. We can find the light’s “view” matrix by taking the inverse of its world transform, the same way we do for a camera.

Since directional lights’ rays are all parallel, we will use our old friend the orthographic projection as our projection transform.

Determining the bounds of the view volume is a notorious problem full of trade-offs. To best utilize the texels in the shadow map, we want to tightly bound the scene with the view volume. If we don’t correctly select the top, left, bottom, and right parameters of the projection, we might not capture portions of the scene in the shadow map, causing us to later erroneously conclude they are in shadow when they should be illuminated (or vice versa). If we don’t select the near and far Z values carefully, we waste the limited precision of the depth range, causing blocky, jagged shadow edges.

For the time being, we will keep things simple and use a fixed projection matrix that happens to work well for our sample scene, but if we were writing a general-purpose renderer, we’d need to design heuristics for shaping the projection matrix for precision and accuracy.

We add a `projectionMatrix` member to our `Light` class to compute the projection transform.





```
var projectionMatrix: simd_float4x4 {
    return simd_float4x4(orthographicProjectionWithLeft: -1.5,
                        top: 1.5,
                        right: 1.5,
                        bottom: -1.5,
                        near: 0,
                        far: 10)
}
```

## Creating Depth Textures

To the `Light` class we will also add a member that indicates if a light is a shadow-casting light, and an optional texture member to store the shadow map itself.

```
var castsShadows = false
var shadowTexture: MTLTexture?
```

Here is the complete configuration of our main directional light:

```
sunLight = Light()
sunLight.type = .directional
sunLight.worldTransform = simd_float4x4(
    lookAt: SIMD3<Float>(0, 0, 0),
    from: SIMD3<Float>(1, 1, 1),
    up: SIMD3<Float>(0, 1, 0))
sunLight.castsShadows = true
```

We use a fixed shadow map resolution of 2048×2048. To create a shadow map, we first populate a Metal texture descriptor with a pixel format of `MTLPixelFormat.depth32Float`, a format in which each texel is a single-precision float depth value.

```
let shadowMapSize = 2048
let textureDescriptor =
    MTLTextureDescriptor.texture2DDescriptor( pixel
        Format: .depth32Float,
        width: shadowMapSize,
        height: shadowMapSize,
        mipmapped: false)
textureDescriptor.storageMode = .private
textureDescriptor.usage = [ .renderTarget, .shaderRead ]
```



We set the storage mode to `MTLStorageMode.private`, which means that the shadow texture will only be allocated on the GPU and not accessible to the CPU. We set its usage to a combination of `MTLTextureUsage.renderTarget` and `MTLTextureUsage.shaderRead`, since we will first render to the shadow map to store the depth values of the scene from the light's perspective, then sample it in the main pass to determine which fragments are illuminated and which are in shadow.

We can then ask the device for a texture and store it in our shadow-casting light:

```
sunLight.shadowTexture = device.makeTexture(descriptor:
textureDescriptor)
```

## ***Multipass Rendering***

So far, we have only encoded one pass per frame, drawing everything directly into a view's drawable texture. However, sometimes we need to render to different textures to achieve certain effects. Shadow mapping is one such effect.

Render passes are delimited by render command encoders. In other words, whenever we want to start a new pass, we must construct a new render pass descriptor and create a new render command encoder. Command buffers can contain any number of render passes, but any time we need to change render attachments (textures), we must start a different pass. We call encoding more than one pass per frame "multipass rendering."

To accommodate multipass rendering, we will refactor our draw method to call one member method per pass, first `drawShadows()`, then

```
drawMainPass() :
```

```
let commandBuffer = commandQueue.makeCommandBuffer() !
drawShadows(light: sunLight, commandBuffer: commandBuffer)
```



```
drawMainPass(renderPassDescriptor: renderPassDescriptor,  
commandBuffer: commandBuffer)  
  
commandBuffer.present(view.currentDrawable!)  
commandBuffer.commit()
```

## ***Render Pass Descriptors for Shadow Mapping***

To perform our shadow mapping pass, we first need to create a render pass descriptor and configure it to store depth values. To do so, we set our light's shadow texture as the texture of the descriptor's *depth attachment*:

```
let renderPassDescriptor = MTLRenderPassDescriptor()  
renderPassDescriptor.depthAttachment.texture = light.shadowTexture
```

To ensure the depth texture is cleared to a known value at the start of the pass, we set its load action to `MTLLoadAction.clear` and set its clear depth to 1.0, signifying an infinite distance.

```
renderPassDescriptor.depthAttachment.loadAction = .clear  
renderPassDescriptor.depthAttachment.clearDepth = 1.0
```

Finally, we set the depth attachment's storage action to

`MTLStorageAction.store` so the results of the shadow pass are available for the main pass.

```
renderPassDescriptor.depthAttachment.storeAction = .store
```

Rendering then proceeds as normal. We construct a model-view-projection for each node from the light's view and projection matrices and the node's world matrix.

## ***Shadow Mapping Vertex Function and Render***

## ***Pipeline***

The shader functions for shadow mapping are remarkably simple. In fact,

there is no fragment function at all: the rasterized depth is written directly to the shadow map and we don't need to write any custom code. The vertex function is trivial: we just multiply the vertex's model position by the composed model-view-projection matrix to produce a clip space position:

```
vertex float4
    vertex_shadow( VertexIn in
        [[stage_in]],
        constant float4x4 &modelViewProjectionMatrix [[buffer(2)]] )
{
    return modelViewProjectionMatrix * float4(in.position, 1.0);
}
```

To turn this vertex function into a render pipeline, we populate a render pipeline descriptor as usual, omitting the fragment function:

```
let shadowRenderPipelineDescriptor = MTLRenderPipelineDescriptor()
shadowRenderPipelineDescriptor.vertexDescriptor = vertexDescriptor
shadowRenderPipelineDescriptor.depthAttachmentPixelFormat =
    .depth32Float
shadowRenderPipelineDescriptor.vertexFunction =
    library.makeFunction(name: "vertex_shadow")

do {
    shadowRenderPipelineState = try
        device.makeRenderPipelineState(descriptor:
            shadowRenderPipelineDescriptor)
} catch {
    fatalError("Error while creating render pipeline state: \
(error) ")
}
```

Note that we can reuse the same vertex descriptor that we use for the main pass; we just don't actually use any attributes other than the model-space position.

## ***Using Shadow Maps***

In the main pass, we will use the shadow map to determine which fragments are in shadow. We do this by first determining the fragment's depth from the light's perspective. If this depth is less than the value stored in the shadow map, the point is illuminated; otherwise it is in shadow.

To construct the fragment's light-space position, we need to add a view-projection matrix to our light constants:

```
struct Light {
    float4x4 viewProjectionMatrix;
    float3 intensity;
    float3 direction;
    LightType type;
};
```

We can then adapt our directional lighting shader code to determine the degree to which we are in shadow and use this to modulate our lighting:

```
float shadowFactor = 1 - shadow(in.worldPosition, shadowMap,
light.viewProjectionMatrix);
// ...
diffuseFactor = shadowFactor * saturate(dot(N, L));
specularFactor = shadowFactor * powr(saturate(dot(N, H)),
specularExponent);
```

The real meat of the algorithm is in the `shadow()` function, so let's step through that, starting with the function signature:

```
static float shadow(float3 worldPosition,
                    depth2d<float, access::sample> depthMap,
                    constant float4x4 &viewProjectionMatrix)
{
```

The world position parameter is the world position of the current fragment, provided by the rasterizer. The depth map parameter is the light's shadow texture, and the view-projection matrix is the light's view-projection matrix.

To determine the depth of the closest fragment to the light, we transform the fragment into the light's clip space, then manually perform a perspective divide to move into NDC. From there, we determine the shadow map texture coordinates that correspond to the fragment:





```
float4 shadowNDC = (viewProjectionMatrix * float4(worldPosition,
1));
shadowNDC.xyz /= shadowNDC.w;
float2 shadowCoords = shadowNDC.xy * 0.5 + 0.5;
shadowCoords.y = 1 - shadowCoords.y;
```

To compare the shadow depth and the fragment depth, we will sample the shadow map at these coordinates. Rather than taking a sampler state, we can use a `constexpr` *sampler* configured specially for depth map comparison:

```
constexpr sampler
    shadowSampler( coord::normalized,
        address::clamp_to_edge,
        filter::linear,
        compare_func::greater_equal);
```

To perform the depth comparison, we use the `sample_compare` function on the depth texture, passing the fragment's shadow map coordinates and the depth value to compare to.

```
float shadowCoverage = depthMap.sample_compare(shadowSampler,
shadowCoords, shadowNDC.z);
return shadowCoverage;
}
```

Since our sampler is configured with the “greater-or-equal” comparison function, the result will be 0 if the fragment is closer to the light than the value in the shadow map, and 1 otherwise. A value of 1 therefore means the fragment is in shadow. We subtract the result from 1 in the vertex shader to get the value by which we modulate the lighting.

At this point, we've implemented shadow mapping. If we run the sample app at this point, though, we see something awful.



## Shadow Acne

The checkered or banded artifacts produced by shadow mapping are called “shadow acne.”

If our shadow map had infinite resolution and precision, our approach so far would probably work just fine. However, since both are limited, it’s possible for the depth comparison to produce false positives, considering fragments in shadow when they should be lit. This happens when the quantized depth sampled from the shadow map is just barely farther than the fragment depth.

The simplest way to address shadow acne is with a *depth bias*. We subtract a tiny value from the projected depth, which causes the edge cases that produce shadow acne to resolve negatively instead of positively for shadowing.

Here is the modified portion of the `shadow` function:

```
const float depthBias = 5e-3f;
float shadowCoverage = depthMap.sample_compare(
    shadowSampler,
    shadowCoords,
    shadowNDC.z - depthBias);
```

Implementing this depth bias resolves the issues for this scene, but the

actual value of the depth bias depends on various factors such as the projection transform and the precision and resolution of the shadow map. If the bias needs tuning, it can be passed as a per-light parameter rather than using a hardcoded constant.

Here is our completed sample app with directional shadows:



In the next article we will look at multisampled antialiasing, a technique for producing smoother images with relatively little processing overhead.

# Day 20: Multisample Antialiasing



Warren Moore ·

7 min read · Apr 29, 2022

*This series of posts is my attempt to present the Metal graphics programming framework in small, bite-sized chunks for Swift app developers who haven't done GPU programming before.*

*If you want to work through this series in order, start [here](#). To download the sample code for this article, go [here](#).*

When rendering, the resolution of our virtual canvas can be an important performance consideration: the more pixels we have to shade, the longer our frames take to process. So far, we have been using `MTKView`'s default behavior, which determines the size of drawables by multiplying the view's bound's size by the layer's contents scale. On a Retina display, this tends to create drawable textures with many millions of pixels.

One option we have to reduce render target memory and increase performance is to render at a lower resolution by manually controlling the drawable size. For example, we might choose to render at a fixed vertical resolution of 1080 pixels and allow the system to “upscale” the result to the appropriate size by bilinear filtering. This manual approach is frequently used by console games, which require consistent framerates and have constrained memory budgets.

The downside to using smaller render targets, of course, is that we can introduce artifacts, such as “jaggies,” visible stairstep patterns that result from not having enough pixels to represent smooth geometric outlines.

One response to such aliasing is *supersampling*, which renders multiple samples per pixel and averages the result to produce a smoother image. We

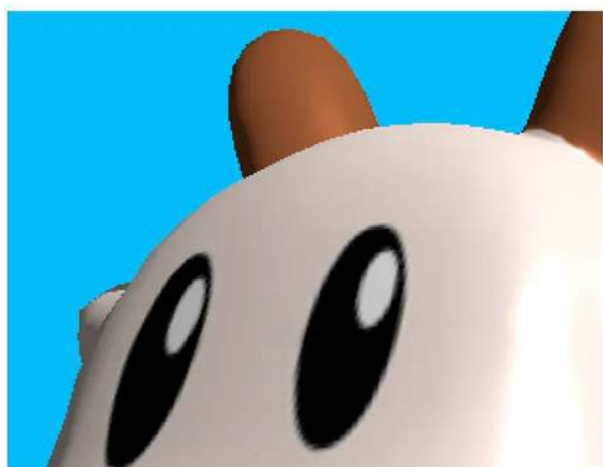
call the number of samples evaluated per pixel the *sample count*. If the total number of samples rendered by reducing the nominal resolution and taking multiple samples per pixel reduces the overall workload, supersampling can be a reasonable choice. However, there is often a better alternative.

## ***Multisample Antialiasing***

*Multisample antialiasing* (MSAA) is an antialiasing technique that exploits a particular fact of geometry to reduce the workload below that of supersampling at an equivalent sample count. Specifically, MSAA takes advantage of the fact that jaggies are most prone to occur on the silhouette edges of primitives. Within a primitive, rasterization tends to produce smooth results, but at primitive edges, a primitive either covers a pixel or it doesn't.

MSAA works by taking a single sample for each pixel inside a primitive and multiple subpixel samples along primitive edges. The samples are stored in a multisample rendering target, which takes up memory proportional to the resolution multiplied by the sample count. MSAA render targets can be much larger than their single-sampled counterparts; the savings come from the intelligent sampling along primitive edges.

Below is an example of the effects of MSAA on a rendered image. Note that in the left image, silhouette edges are quite jagged, while in the the right image, which uses 4 MSAA samples per pixel, the edges are much more smoothly blended.



**No MSAA**



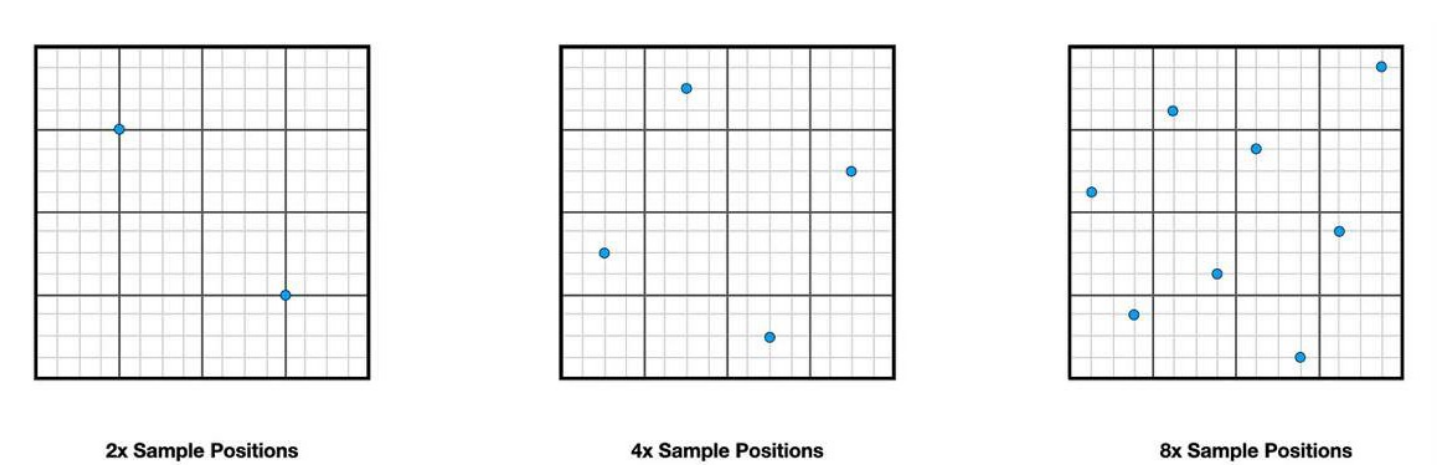
**MSAA 4x**

In order to turn a multisampled render texture into an image that can be displayed, we must *resolve* it into a single-sampled texture. This process happens at the end of a pass after rendering is complete, and is configured on the render pass descriptor.

## Sample Counts and Positions

Increasing the sample count tends to increase image quality; that’s why we’re using MSAA in the first place. But each additional sample also occupies additional texture memory (at least on some GPUs). Additionally, most GPUs support a fixed maximum sample count. In Metal, the supported sample counts tend to be powers of 2: 1, 2, 4, and 8.

Each sample count has a corresponding default arrangement of the subpixel sample positions. The sample positions for 2x, 4x, and 8x MSAA are shown in the figure below.



## MSAA with MTKView

The `MTKView` class is MSAA-aware: you can easily configure it to create MSAA render targets and resolve them automatically at the end of the render pass that renders into the view’s drawable.

You configure the view for MSAA by setting its `sampleCount` property. A common sample count is 4.

```
mtkView.sampleCount = 4
```





Setting this single property causes the `MTKView` to allocate MSAA-enabled color and depth textures on your behalf, and configure the render pass descriptors returned by the `currentRenderPassDescriptor` property to perform MSAA.

The only other change you are obligated to make to support MSAA is to inform the render pipeline creation process that you will be using MSAA. This is done by setting the `rasterSampleCount` property on the render pipeline descriptor:

```
let renderPipelineDescriptor = MTLRenderPipelineDescriptor()
renderPipelineDescriptor.rasterSampleCount = mtkView.sampleCount
```

Now, when rendering, the graphics pipeline will run the fragment shader multiple times per pixel, store the resulting colors in the multisampled textures created by the view, and resolve the MSAA textures into their single-sampled counterparts (including the drawable texture, which is then presented as usual).

## ***Creating MSAA Render Targets Manually***

Of course, this article series isn't just about using MetalKit; it's also about using Metal's lower-level facilities. So let's look at how to manually create render target textures and configure render pass descriptors to use them.

We add a few properties to our renderer class to support MSAA rendering:

```
let rasterSampleCount = 4
var msaaColorTexture: MTLTexture?
var msaaDepthTexture: MTLTexture?
```

All textures have a fixed size, so if and when the drawable size of the view changes, we need to throw away any existing MSAA targets and recreate them. Fortunately, we can respond to such changes in the

mtkView(\_:drawableSizeWillChange:) delegate method:

```
func mtkView(_ view: MTKView, drawableSizeWillChange size: CGSize)
{
    msaaColorTexture = nil
    msaaDepthTexture = nil
}
```

To recreate these textures on-demand, we add a method called

`makeMSAARenderTargetsIfNeeded()` that is called at the top of the draw method.

First of all, we don't need to create MSAA targets if MSAA isn't being used. We check this by testing whether the renderer's sample count is greater than 1:

```
func makeMSAARenderTargetsIfNeeded()
{
    if rasterSampleCount == 1 { return
    }
```

We only need to recreate render targets if (1) they haven't been created previously, (2) their size differs from our drawable size, or (3) their sample count differs from our desired sample count.

We first convert the drawable width and height to integers:

```
let drawableWidth = Int(view.drawableSize.width)
let drawableHeight = Int(view.drawableSize.height)
```

Now we test our existing MSAA color target, if any, for the conditions listed above.

```
if msaaColorTexture == nil ||
    msaaColorTexture?.width != drawableWidth ||
    msaaColorTexture?.height != drawableHeight ||
    msaaColorTexture?.sampleCount != rasterSampleCount
```

In the event we do need to create MSAA targets, we use the now-familiar

procedure of filling out a texture descriptor:

```
let textureDescriptor = MTLTextureDescriptor()
textureDescriptor.textureType = .type2DMultisample
textureDescriptor.sampleCount = rasterSampleCount
textureDescriptor.pixelFormat = view.colorPixelFormat
textureDescriptor.width = drawableWidth
textureDescriptor.height = drawableHeight
textureDescriptor.storageMode = .private
textureDescriptor.usage = .renderTarget
```

The chief differences between this descriptor and others we have seen are the texture type, which is set to `MTLTextureType.type2DMultisample`, and the `sampleCount` property, which is set to match the preferred MSAA sample count. Since the texture will act as a render target and does not need to be accessible to the CPU, we set its usage and storage mode accordingly.

Then we can create and store the MSAA color texture:

```
msaaColorTexture = device.makeTexture(descriptor: textureDescriptor)
```

The procedure for creating the depth texture is almost identical, so I won't repeat it here. Instead of using the view's `colorPixelFormat` property to select the depth pixel format, we use its `depthStencilPixelFormat`.

## ***Creating MSAA Render Pass Descriptors***

Since we are now creating MSAA targets manually, we do not need to inform `MTKView` that we are using MSAA; we can leave the view's `sampleCount` set to 1. However, this means we are responsible for creating our own render pass descriptors, since the one returned by `currentRenderPassDescriptor` will be invalid for MSAA rendering.

We will write a method, `renderPassDescriptor(colorTexture:,depthTexture:)` that handles both the MSAA and non-MSAA cases.

First, we instantiate the render pass descriptor:

```
func renderPassDescriptor(colorTexture: MTLTexture,
                          depthTexture: MTLTexture?)
    -> MTLRenderPassDescriptor
{
    let renderPassDescriptor = MTLRenderPassDescriptor()
```

For each render target that has an MSAA target, we need to configure the corresponding render pass attachment with the MSAA texture and a “resolve texture” to store the final results.

We begin by checking whether MSAA is enabled and setting up the MSAA targets if so:

```
let msaaEnabled = (rasterSampleCount > 1)

if msaaEnabled
{
    renderPassDescriptor.colorAttachments[0].texture =
msaaColorTexture
    renderPassDescriptor.colorAttachments[0].resolveTexture =
colorTexture
    renderPassDescriptor.colorAttachments[0].loadAction = .clear
    renderPassDescriptor.colorAttachments[0].clearColor =
view.clearColor
    renderPassDescriptor.colorAttachments[0].storeAction =
.multisampleResolve

    renderPassDescriptor.depthAttachment.texture = msaaDepthTexture
    renderPassDescriptor.depthAttachment.resolveTexture =
depthTexture
    renderPassDescriptor.depthAttachment.loadAction = .clear
    renderPassDescriptor.depthAttachment.clearDepth =
view.clearDepth
    renderPassDescriptor.depthAttachment.storeAction =
.multisampleResolve
}
```

The thing to pay attention to here is the store action. For both the color and depth attachment, the store action is `MTLStoreAction.multisampleResolve`, which tells Metal to average the multiple samples for each pixel down to a single value to write into the resolve texture.

The non-MSAA case is easier to handle and should look familiar from our explorations with shadow mapping (the only exception being that we discard the contents of the depth map at the end of the pass):





```

else {
    renderPassDescriptor.colorAttachments[0].texture = colorTexture
    renderPassDescriptor.colorAttachments[0].loadAction = .clear
    renderPassDescriptor.colorAttachments[0].clearColor =
view.clearColor
    renderPassDescriptor.colorAttachments[0].storeAction = .store

    renderPassDescriptor.depthAttachment.texture = depthTexture
    renderPassDescriptor.depthAttachment.loadAction = .clear
    renderPassDescriptor.depthAttachment.clearDepth =
view.clearDepth
    renderPassDescriptor.depthAttachment.storeAction = .dontCare
}

```

Finally, we return the configured render pass descriptor for use in our draw method:

```

return renderPassDescriptor
}

```

We can now use this method instead of asking the view for a render pass descriptor. We still use the drawable's texture and the view's internal depth texture; no reason to reinvent that particular wheel.

```

guard let drawable = view.currentDrawable else { return }
let renderPassDescriptor = renderPassDescriptor(
    colorTexture: drawable.texture,
    depthTexture: view.depthStencilTexture)

```

The difference afforded by MSAA may seem subtle, but at lower base resolutions, it can make all the difference between swimming jaggies and smooth sailing. Here's a multisample antialiased version of the screenshot from last time:



Next time, we will look at point lights, to add more expressiveness to our lighting environments.

# Day 21: Point Lights



Warren Moore ·

4 min read · May 1, 2022

*This series of posts is my attempt to present the Metal graphics programming framework in small, bite-sized chunks for Swift app developers who haven't done GPU programming before.*

*If you want to work through this series in order, start [here](#). To download the sample code for this article, go [here](#).*

Directional lights are great for covering large scenes with powerful light from a dominant direction, but apart from the sun, we don't see many directional lights in the real world. We need another type of light to model light sources that radiate in all directions and can move around the scene. These lights are called points lights, or punctual lights, to emphasize the fact that they have a particular position and emit light radially from that position.



In this article, we will discuss how to update our lighting shaders to consider contributions from points lights. Along the way, we will introduce an important physical phenomenon that affects how light intensity diminishes over distance, called *attenuation*.

Since we already updated our `Light` class with a transform that allows us to express a light's position, we do not need to modify it any further. We will add a new member to our light type enumeration, called `.omni`, to emphasize that points lights radiate in all directions (i.e., they are omnidirectional).

```
enum LightType : UInt32
{
    case ambient
    case directional
    case omni
}
```

We make a similar change to the `LightType` enum in our shader code as well. To make lighting calculations easier, we also extent our `LightConstants` structure with a `position` member, which holds the world-space position of the light:

```
struct Light {
    float4x4 viewProjectionMatrix;
    float3 intensity;
    float3 position; // world-space position
    float3 direction; // view-space direction
    LightType type;
};
```

## ***Attenuation***

Directional lights are presumed to be infinitely distant, which implies that their intensity does not decrease (“fall off”) with distance. Point lights, on the other hand, are assumed to be local, which means their intensity should fall off proportional to the squared distance to a surface.

This distance-related falloff is one kind of *attenuation*, or reduction in

intensity. To capture it mathematically, we need to know the distance from the light to the surface. We then multiply the light's intensity by the reciprocal of the distance squared.

## Implementing Attenuation

To implement attenuation in a shader, we first find the vector pointing from the light's world position to the current fragment's world position (call it `toLight`). This vector points in the same direction as the `L` vector we have used previously, but it is *not* normalized.

Because we know that the dot product of two vectors is equal to the product of their lengths multiplied by the cosine of the angle between them, and because we know that the angle between a vector and itself is 0, we can find the squared distance to the light by taking the dot product of `toLight` with itself:

```
lightDistSq = dot(toLight, toLight)
```

Dividing 1 by the squared distance gives the attenuation factor, but there is a problem. When the distance goes to zero (i.e., when the light is “on” the surface), the reciprocal goes to infinity. We hack around this by taking the reciprocal of the max of the squared distance and a small value, preventing division by zero.

```
attenuation = 1 / max(lightDistSq, 1e-4)
```

We can combine these expressions into a function that yields the distance attenuation factor of a light, given a reference to the light and the vector from the surface to the light:

```
float distanceAttenuation(constant Light &light, float3 toLight)
{ switch (light.type) {
  case LightTypeOmnidirectional: {
    float lightDistSq = dot(toLight, toLight);
```



```

        return 1.0f / max(lightDistSq, 1e-4);
        break;
    }
    default:
        return 1.0;
}
}

```

We then implement the rest of the lighting model in a very similar way to our directional implementation. The chief difference is that we normalize the `toLight` vector to find our light direction, rather than using the light's own direction vector.

Here is the complete code for our point light lighting implementation:

```

case LightTypeOmnidirectional: {
    float3 toLight = (light.position - in.worldPosition);
    float attenuation = distanceAttenuation(light, toLight);

    float3 L = normalize(toLight);
    float3 H = normalize(L + V);
    diffuseFactor = attenuation * saturate(dot(N, L));
    specularFactor = attenuation * powr(saturate(dot(N, H)),
specularExponent);
    break;
}

```

## ***Point Lights in Action***

To demonstrate our new point lights, we add a few of them to our scene:

```

let orbitLight0 = Light()
orbitLight0.color = SIMD3<Float>(0.45, 0, 0.95)
orbitLight0.intensity = 0.8
orbitLight0.type = .omni

let orbitLight1 = Light()
orbitLight1.color = SIMD3<Float>(0.95, 0.45, 0)
orbitLight1.intensity = 0.8
orbitLight1.type = .omni

let orbitLight2 = Light()
orbitLight2.color = SIMD3<Float>(0, 0.95, 0.45)
orbitLight2.intensity = 0.8
orbitLight2.type = .omni

orbitLights = [orbitLight0, orbitLight1, orbitLight2]

```





We then animate their positions around our Spot the cow model to demonstrate fully dynamic point lighting:

```
let orbitRadius: Float = 1.0
for i in 0..
```

With these changes made, we can run the sample app and see our orbiting point lights in action:



Next time, we will turn our attention to efficiently rendering many objects with GPU instancing.

# Day 22: Instancing



Warren Moore ·

6 min read · May 2, 2022

*This series of posts is my attempt to present the Metal graphics programming framework in small, bite-sized chunks for Swift app developers who haven't done GPU programming before.*

*If you want to work through this series in order, start [here](#). To download the sample code for this article, go [here](#).*

So far, there has been a one-to-one relationship between nodes and draw calls: we encode one draw call per mesh (or really, submesh). Suppose we want to render the same mesh many times. We could certainly write a loop that encodes one draw call for each time we want to render the mesh. However, encoding isn't a cheap operation, so reducing draw call count is desirable.

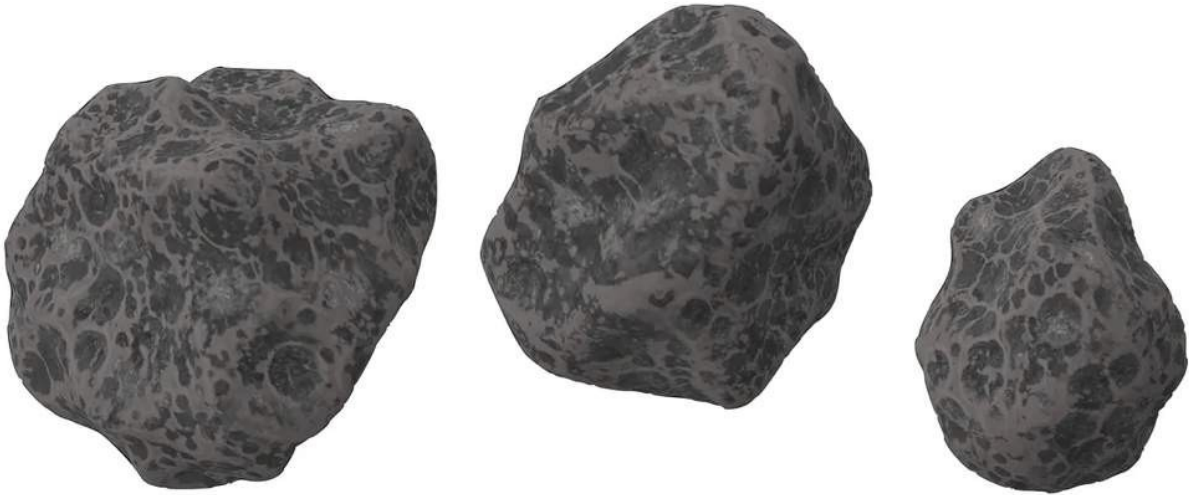
It turns out Metal supports a feature that allows us to draw many meshes with a single draw call: *instancing*. With instancing, we tell Metal how many times to draw a mesh, while providing per-instance data in a buffer.

This per-instance data is just like the per-node constant data we have used in our shaders previously. The only thing that changes is how we access it.

In the following sections, we will see how to organize batches of nodes so they can be rendered with instanced draw calls, increasing efficiency while allowing us to vary the animated transformations of each instance independently.

## ***The Sample Scene***

We will build an asteroid field as our sample scene to demonstrate instancing. The background will be a skybox textured with distant stars, while the foreground will consist of about 100 gently tumbling asteroids. Each asteroid uses one of three meshes:



Each asteroid has a unique position, scale, and rotation parameters. Asteroids are batched together so that only one instanced draw call per mesh is required.

## ***Loading Mesh Assets***

Our asteroid mesh assets are stored in three files: `asteroid1.obj`, `asteroid2.obj`, and `asteroid3.obj`. This naming scheme allows us to iterate over the file names using string interpolation:

```
for asteroidID in 1...3 {
  let assetURL =
    Bundle.main.url( forResource: "asteroid\
(asteroidID)", withExtension: "obj")

  let mdlAsset =
    MDLAsset( url:
      assetURL,
      vertexDescriptor: mdlVertexDescriptor,
      bufferAllocator: bufferAllocator)
```

The remainder of the code for loading the asset's mesh and converting it to an `MTKMesh` remains the same.

***Creating Node Batches***

For each asteroid mesh, we will create many asteroid nodes that are grouped into a *batch*. A batch is a set of nodes that share a mesh and texture while having varying transforms.

We will store each asteroid's state separate from its node, in an `AsteroidState` object. This allows us to define custom properties such as the asteroid's rotation axis and angular velocity (rotation rate). We then derive the corresponding node's transform from these properties.

```
class AsteroidState {
    var position = SIMD3<Float>(0, 0, 0)
    var scale: Float = 1.0
    var rotationAxis = SIMD3<Float>(0, 1, 0)
    var rotationAngle: Float = 0.0
    var angularVelocity: Float = 0.0
}
```

We define a few properties on our renderer to hold our batches, our asteroid nodes, and our asteroid state objects:

```
var batches = [[Node]]()
var asteroidNodes = [Node]()
var asteroids = [AsteroidState]()
```

For each mesh, we want to create a batch of asteroids, each of which has unique, randomly generated properties. Specifically, we generate a random position, a random rotation axis, a random rotation rate, and a random scale.

```
let asteroidsPerMesh = 50
var batch = [Node]()
for _ in 0..
```



```
asteroid.scale = Float.random(in: 0.6.....1.2)

asteroids.append(asteroid)
```

At the same time, we create a node for each asteroid and add it to the current batch, as well as the list of asteroids, which we use later when animating the asteroids.

```
let node = Node(mesh: mesh)
  node.texture = texture
  asteroidNodes.append(node)
  batch.append(node)
}
```

Once we are done generating a batch of asteroids, we add it to our collection of batches, which we will iterate over when rendering.

```
batches.append(batch)
```

## ***Updating Per-Instance Constants***

Updating the per-instance data is similar to how we have previously updated per-node constants. First, we animate the asteroids according to their individual properties. Then, we copy the updated transforms into the constant buffer.

Updating the node state consists of incrementally rotating each asteroid around its randomized axis of rotation according to its rotational velocity, then building its transformation from its translation, scale, and rotation properties:

```
for (asteroidIndex, asteroidState) in asteroids.enumerated()
{
  asteroidState.rotationAngle +=
    asteroidState.angularVelocity * timestep

  let S = simd_float4x4(scale:
    SIMD3<Float>(repeating: asteroidState.scale))

  let R = simd_float4x4(rotateAbout: asteroidState.rotationAxis,
```





```

        byAngle: asteroidState.rotationAngle)

    let T = simd_float4x4(translate: asteroidState.position)

    asteroidNodes[asteroidIndex].transform = T * R * S
}

```

We need to store a constant buffer offset for each batch. Before allocating constant storage for the batches, we first clear the offsets from the previous frame.

```
batchConstantsOffsets.removeAll()
```

Then, we iterate over the batches, copying the model-to-world transformations of each of the batch's nodes into the constant buffer.

```

for batch in batches {
    let layout = MemoryLayout<InstanceConstants>.self
    let offset = allocateConstantStorage(
        size: layout.stride * batch.count,
        alignment: layout.stride)

    let instanceConstants = constantBuffer.contents()
        .advanced(by: offset)
        .bindMemory(to: InstanceConstants.self,
                    capacity: batch.count)

    for (nodeIndex, node) in batch.enumerated()
    { instanceConstants[nodeIndex] =
        InstanceConstants(modelMatrix: node.worldTransform)
    }

    batchConstantsOffsets.append(offset)
}

```

## ***Instanting in the Vertex Function***

We will use the same type of constant data for each instance that we used for each node previously: the model-to-world transformation matrix. To emphasize that this data will be fetched per-instance, we name the structure `InstanceConstants`:

```
struct InstanceConstants {
```

```
float4x4 modelMatrix;
};
```

Now, rather than taking a reference to a single `NodeConstants` structure, we take a pointer to an array of `InstanceConstants` structures, along with the built-in instance ID:

```
vertex VertexOut
vertex_main( VertexIn in
[[stage_in]],
constant InstanceConstants *instances [[buffer(2)]],
constant FrameConstants &frame [[buffer(3)]],
uint instanceID [[instance_id]])
{
```

Taking a pointer rather than a reference allows us to index into the instance constants buffer to retrieve the constants for a single instance:

```
constant InstanceConstants &instance = instances[instanceID];
```

We can then use the instance's model matrix just as we have been using the node's model matrix:

```
float4x4 modelViewMatrix = frame.viewMatrix * instance.modelMatrix;
```

The remainder of the vertex function is unchanged.

## ***Instanced Draw Calls***

The heart of our instanced drawing routine is the

```
drawIndexedPrimitives(type:
indexCount:indexType:indexBuffer:indexBufferOffset: instanceCount:)
```

. This tells the encoder to encode an instanced, indexed draw call, which will

draw `instanceCount` instances of whatever mesh is being rendered.

To draw each batch of asteroids, we iterate over our array of batch arrays.

For each batch, we select the first node and mesh as “representative,” and use them to bind the resources that will be used to draw the mesh.

```
for (batchIndex, batch) in batches.enumerated()
{ guard let node = batch.first else { continue }
  guard let mesh = node.mesh else { continue }
```

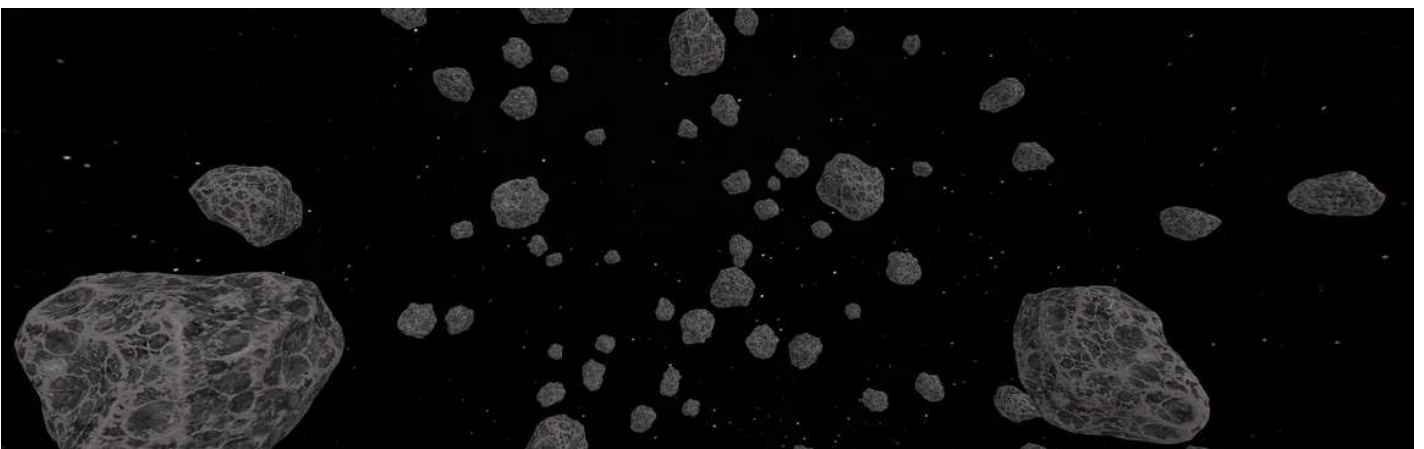
We also need to bind the instance constants, which we do by using the previously calculated per-batch constants offsets.

```
renderCommandEncoder.setVertexBuffer( constantBuffer,
offset: batchConstantsOffsets[batchIndex],
index: 2)
```

The rest of our `MTKMesh` rendering code is unchanged, apart from the addition of the `instanceCount` parameter:

```
renderCommandEncoder.drawIndexedPrimitives( type: submesh.primitiveType,
indexCount: submesh.indexCount,
indexType: submesh.indexType,
indexBuffer: indexBuffer.buffer,
indexBufferOffset: indexBuffer.offset,
instanceCount: batch.count)
```

Running the sample app shows our many, diverse asteroids a-tumbling in the depths of space:





Now that we have a dense, perilous field of asteroids, it would be a shame not to be able to fly through them. In the next article, we will introduce interactivity and camera controllers.

# Day 23: Interaction



Warren Moore ·

10 min read · May 3, 2022

*This series of posts is my attempt to present the Metal graphics programming framework in small, bite-sized chunks for Swift app developers who haven't done GPU programming before.*

*If you want to work through this series in order, start [here](#). To download the sample code for this article, go [here](#).*

What fun is a dense asteroid field in the middle of space if you can't fly through it? In this article, we will learn the basics of interaction. We will learn how to map mouse movement and keyboard presses in macOS to camera movement, and we will also get familiar with the GameController framework and learn how to simulate a game controller on iOS even if we don't have a physical controller.

## Camera Controllers

So far, to establish a point of view for our virtual camera, we have set a fixed position and orientation. Implementing an interactive camera consists of computing a new position and orientation for the camera when the user provides input.

A *camera controller* is an object that has the responsibility of turning input events into camera transform updates. There are many possible camera controller schemes which can be selected from depending on the needs of the application. A few are described below.

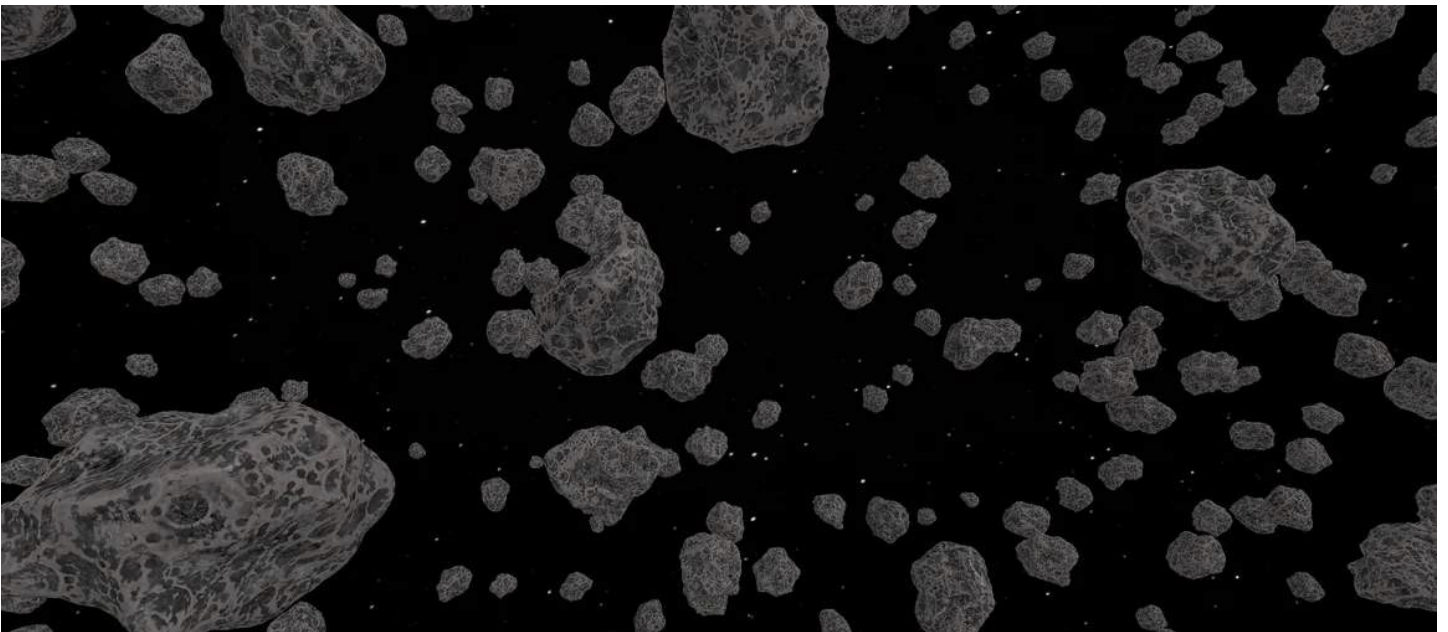
- **Turntable:** The camera points toward the center of the scene, which can be rotated about its vertical axis, as if it were sitting on a turntable or



potter's wheel.

- **Arcball:** The scene is enclosed in an invisible sphere (like a hamster ball), and rotates about an axis perpendicular to the direction of drag gestures.
- **Pan:** The camera moves in the XY (vertical) plane at a constant distance
- **Truck:** The camera moves in the XZ plane at a constant height
- **Fly:** The camera moves freely in 3D space, often with the keyboard controlling position and the mouse controlling orientation.

In the sections below we will discuss how to implement a fly camera so we can fly around the instanced asteroid field we made last time.



## ***Fly Cameras***

A fly camera can have up to six *degrees of freedom*: movement along the X, Y, and Z axes, and rotation around each axis. Rotation around the X, Y, and Z axes is called *pitch*, *yaw*, and *roll*, respectively. Other texts use different a different order of axes, but the motion is the same.

The fly camera controller we will implement will only have *four* degrees of freedom: we will not allow the camera to roll around its local Z axis, nor will we allow translational movement along the local Y axis. Even with these limitations, our camera controller will allow us to navigate quite naturally in 3D space.

We will implement the logic for our fly camera controller in the

`FlyCameraController` class. The controller will hold a reference to a node, called its point of view, that represents the virtual camera:

```
class FlyCameraController {
    let pointOfView: Node

    //...

    init(pointOfView: Node)
    { self.pointOfView =
      pointOfView
    }

    //...
}
```

The purpose of the camera controller is to turn input events into camera motion. The controller stores a few properties that it composes into the camera transform. The `eye` member stores the camera position, the

`look` member stores the forward direction vector, and the `up` member stores the current up direction vector. It is no accident that these members look a lot like the parameters to the `lookAt` transform initializer.

```
private var eye = SIMD3<Float>(0, 0, 0)
private var look = SIMD3<Float>(0, 0, -1)
private var up = SIMD3<Float>(0, 1, 0)
```

We also need a few constants that affect camera behavior. `invertYLook` is a boolean that controls whether the pitch angle is inverted before rotation is performed, a common option in video games. `eyeSpeed` holds the speed, in units per second, at which the camera moves. `radiansPerLookPoint` holds the number of radians subtended by one screen point, which determines rotation speed.

```
let invertYLook = false
let eyeSpeed: Float = 6
let radiansPerLookPoint: Float = 0.017
```

The central method of the camera controller is

```
update(timestep:lookDelta:moveDelta:):
```

```
func update(timestep: Float,  
            lookDelta: SIMD2<Float>,  
            moveDelta: SIMD2<Float>)
```

This method is agnostic to input method: we can map keyboard and mouse events, touch gestures, or game controller inputs to these parameters. We will see below how to do so on macOS and iOS.

The update method has three phases: update the position, update the orientation, and assign the new transformation to the point of view.

To find the new position, we first construct normalized vectors pointing in the X and Z directions, since the camera only moves in its XZ plane. We call these directions

`right` and `forward`:

```
let right = normalize(cross(look, up))  
var forward = look
```

We then find the camera's movement direction by multiplying the movement inputs by their respective directions:

```
let deltaX = moveDelta[0], deltaZ = moveDelta[1]  
let movementDir = SIMD3<Float>(  
    deltaX * right.x + deltaZ * forward.x,  
    deltaX * right.y + deltaZ * forward.y,  
    deltaX * right.z + deltaZ * forward.z)
```

Finally, we update the position by multiplying this movement direction by the speed and timestep factors:

```
eye += movementDir * eyeSpeed * timestep
```



To update the orientation, we perform two rotations on the forward direction: one around the `up` vector (yaw), and one around the `right` vector (pitch). Rather than using matrix multiplication, we will exploit the power of quaternions to perform these rotations efficiently.

To find the yaw rotation, we scale the X look input by the rotational speed factor, then construct a quaternion that represents a rotation by this angle around the Y axis:

```
let yaw = -lookDelta.x * radiansPerLookPoint
let yawRotation = simd_quaternion(yaw, up)
```

Finding the pitch rotation is similar, but we optionally apply look-inversion:

```
var pitch = lookDelta.y * radiansPerLookPoint
if (invertYLook) { pitch *= -1.0 }
let pitchRotation = simd_quaternion(pitch, right)
```

We form the combined rotation by multiplying the quaternions together, then update the forward vector by rotating it to the new bearing:

```
let rotation = simd_mul(pitchRotation, yawRotation)
forward = rotation.rotate(forward)
```

Since this procedure has potentially made our forward direction nonorthogonal with the up and right directions, we normalize the forward vector and reconstruct a new up vector:

```
look = normalize(forward)
up = cross(right, look)
```

After this sequence of operations, we are assured that our `right`, `up`, and `forward` vectors form an orthonormal set. Now we simply call on our look-

at initializer to construct the camera's new transformation matrix:

```
pointOfView.transform =  
    simd_float4x4( lookAt: eye + look,  
        from: eye, up: up)
```

That completes the camera controller logic, but how do we determine the input parameters of the update method? Below, we will talk about the different input modes we have available on macOS and iOS.

## ***Mouse Input on macOS***

To determine how far the mouse has moved between updates, we will add two member variables to our view controller, which will drive our camera controller updates:

```
var previousMousePoint = CGPoint.zero  
var currentMousePoint = CGPoint.zero
```

We are mostly concerned with *drag* events, which occur when the mouse is moved while its button is pressed. We implement three methods from the

`NSResponder` class to receive events when the button is pressed, when the mouse is dragged, and when the button is released:

```
override func mouseDown(with event: NSEvent) {  
    let mouseLocation = self.view.convert(event.locationInWindow,  
        from: nil)  
    currentMousePoint = mouseLocation  
    previousMousePoint = mouseLocation  
}  
  
override func mouseDragged(with event: NSEvent) {  
    let mouseLocation = self.view.convert(event.locationInWindow,  
        from: nil)  
    previousMousePoint = currentMousePoint  
    currentMousePoint = mouseLocation  
}  
  
override func mouseUp(with event: NSEvent) {  
    let mouseLocation = self.view.convert(event.locationInWindow,  
        from: nil)  
    previousMousePoint = mouseLocation  
    currentMousePoint = mouseLocation  
}
```





In each event responder method, we convert the mouse location into the 2D coordinates of our view, then update our member variables to track the mouse over time.

## ***Keyboard Input on macOS***

Receiving keyboard events is slightly more involved, since our view controller must be added to the *responder chain* in order to get keypresses. We implement two methods in our view controller to enable this: when the view appears, we ask the window to make the view controller the first responder, then we respond affirmatively when the window system asks us if we want to be first responder:

```
override func viewDidLoad()
{ view.window?.makeFirstResponder(self)
}

override func becomeFirstResponder() -> Bool
{ return true
}
```

To keep track of which keys are pressed at a given moment, we store an array of booleans that hold the pressed state of each possible key code:

```
var keysPressed = [Bool](repeating: false,
                          count: Int(UInt16.max))
```

When a key is pressed or released, we receive the event with two additional responder methods:

```
override func keyDown(with event: NSEvent)
{ keysPressed[Int(event.keyCode)] = true
}

override func keyUp(with event: NSEvent)
{ keysPressed[Int(event.keyCode)] = false
}
```

With the mouse and key events handled, we can now turn to how we map them to our camera controller inputs.

## ***Mapping Keyboard and Mouse Inputs***

To update our camera on a regular cadence, we create a timer that fires at our expected frame-rate of 60Hz. Each time the timer fires, we call our view controller’s `updateCamera(:)` method.

```
let frameDuration = 1.0 /  
                    Double(mtkView.preferredFramesPerSecond)  
Timer.scheduledTimer(withTimeInterval: frameDuration,  
                      repeats: true)  
{ [weak self] _ in  
    self?.updateCamera(Float(frameDuration))  
}
```

In `updateCamera(:)`, we use the mouse and key state accumulated since the previous frame. First, we find the distance the mouse moved, then save the current mouse position as the previous mouse position.

```
let cursorDeltaX = Float(currentMousePoint.x -  
                          previousMousePoint.x)  
let cursorDeltaY = Float(currentMousePoint.y -  
                          previousMousePoint.y)  
previousMousePoint = currentMousePoint  
let mouseDelta = SIMD2<Float>(cursorDeltaX, cursorDeltaY)
```

We will use the traditional first-person shooter key layout (“WASD”) to move the camera forward, backward, left, and right.

```
let forwardPressed = keysPressed[VirtualKey.ANSI_W.rawValue]  
let backwardPressed = keysPressed[VirtualKey.ANSI_S.rawValue]  
let leftPressed = keysPressed[VirtualKey.ANSI_A.rawValue]  
let rightPressed = keysPressed[VirtualKey.ANSI_D.rawValue]
```

We find the net motion by adding up the influences of the pressed keys.

```
let deltaX: Float = (leftPressed ? -1.0 : 0.0) +
                    (rightPressed ? 1.0 : 0.0)
let deltaZ: Float = (backwardPressed ? -1.0 : 0.0) +
                    (forwardPressed ? 1.0 : 0.0)
let keyDelta = SIMD2<Float>(deltaX, deltaZ)
```

Then, we call the camera controller’s update method with our look and movement vectors:

```
cameraController.update(timestep: timestep,
                       lookDelta: mouseDelta,
                       moveDelta: keyDelta)
```

This completes the logic for mouse and keyboard control on macOS. Next, we will discuss how to use the GameController framework to handle physical and virtual game controller inputs.

## ***The GameController Framework***

The GameController framework is a library that allows us to easily receive input from a huge variety of game input devices, from the Apple TV Siri Remote to the Sony DualSense controller. These various devices are abstracted over by the `GCController` class.

To use the GameController framework, we import its module at the top of our view controller’s implementation file

```
import GameController
```

To receive notifications when controllers are connected and disconnected, we register ourselves with the default notification center.

```
private func registerControllerObservers()
{ NotificationCenter.default.addObserver(
    forName: NSNotification.Name.GCControllerDidConnect,
    object: nil,
    queue: nil)
```



```

    { [weak self] notification in
        if let controller = notification.object as? GameController
        { self?.controllerDidConnect(controller)
        }
    }

NotificationCenter.default.addObserver(
    forName: NSNotification.Name.GCControllerDidDisconnect,
    object: nil,
    queue: nil)
{ [weak self] notification in
    if let controller = notification.object as? GameController
    { self?.controllerDidDisconnect(controller)
    }
}
}

```

We also store a reference to the connected controller, if any:

```
var gameController: GameController?
```

We can now listen to game controller connection and disconnection notifications and respond appropriately:

```

private func controllerDidConnect(_ controller: GameController)
{ gameController = controller
}

private func controllerDidDisconnect(_ controller: GameController)
{ gameController = nil
}

```

When we have a game controller connected on macOS, we bypass the keyboard and mouse logic and instead defer to the controller input. A

`GameController` object has an optional member of type `GCExtendedGamepad` that gives us access to the various buttons and directional controls on the controller.

To update the camera controller with gamepad input, we first ask the controller for its gamepad:

```
if let gamepad = gameController?.extendedGamepad {
```



An extended gamepad has built-in support for multiple directional joysticks: these are represented by the `leftThumbstick` and `rightThumbstick` members. Each thumbstick has X and Y axes that smoothly vary based on how far the thumbstick is pressed in each direction. We map the left thumbstick to camera movement and the right thumbstick to camera orientation:

```
let lookX = gamepad.rightThumbstick.xAxis.value
let lookZ = gamepad.rightThumbstick.yAxis.value
let lookDelta = SIMD2<Float>(lookX, lookZ)

let moveZ = gamepad.leftThumbstick.yAxis.value
let moveDelta = SIMD2<Float>(0, moveZ)
```

Then, as we did with our keyboard and mouse input, we update the camera controller with our mapped values:

```
cameraController.update(timestep: timestep,
                        lookDelta: lookDelta,
                        moveDelta: moveDelta)
```

This is all fine, but what if we are running on iOS and don't have a physical game controller? It turns out that the GameController framework once again comes to the rescue.

## ***Virtual Controllers on iOS***

Starting with iOS 15, the GameController framework includes the `GCVirtualController` class, a flexible means of defining a virtual gamepad that renders on the screen and generates gamepad events from multitouch inputs.

To create and connect a virtual controller, we first make a `GCVirtualController.Configuration` object. This allows us to request which sticks and buttons we want our virtual controller to have. iOS determines how to lay out and render each input element. In our case, we just want two

virtual thumbsticks:

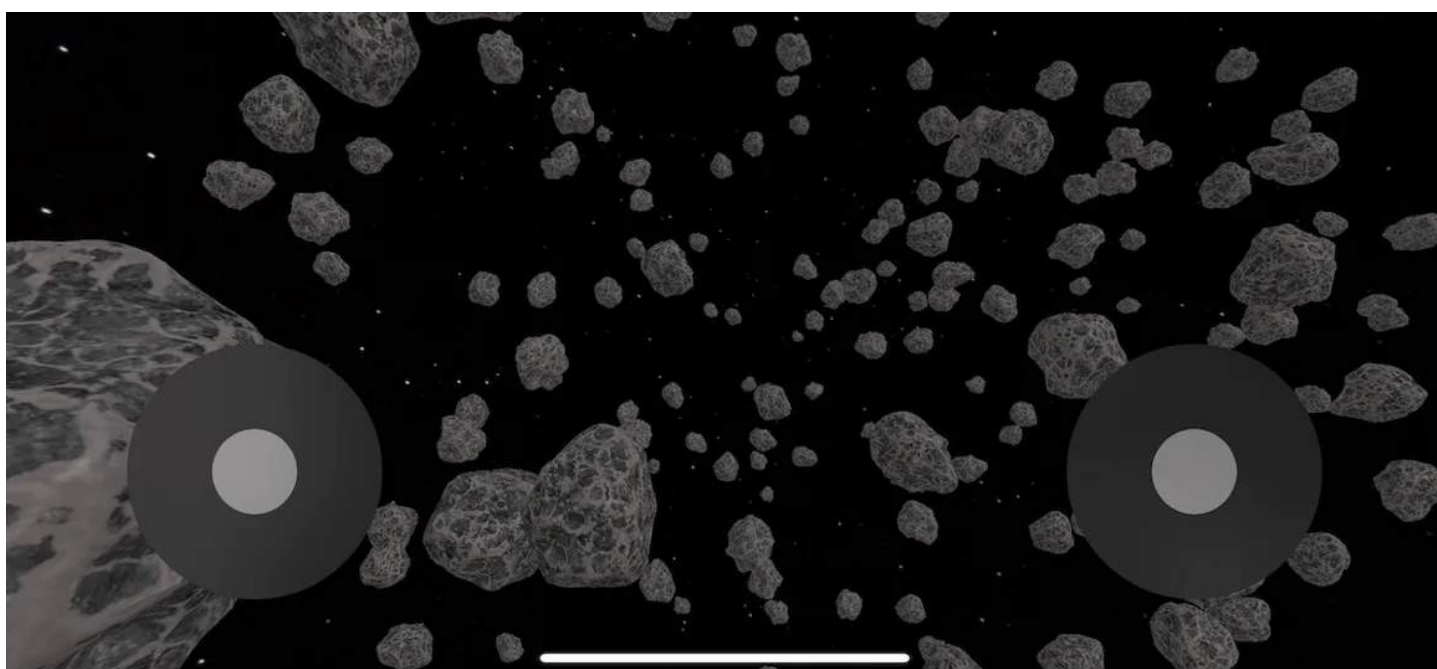
```
let controllerConfig = GCVirtualController.Configuration()
controllerConfig.elements = [
    GCInputLeftThumbstick,
    GCInputRightThumbstick,
]
```

With our controller configuration specified, we can create and connect a virtual controller:

```
let controller =
    GCVirtualController( configuration:
        controllerConfig)
controller.connect()
```

This virtual controller looks to the rest of the system just like a physical controller. As soon as we call `connect()`, the virtual controller appears on- screen, and a connection notification is sent to our view controller, allowing us to poll for controller status exactly as if we'd plugged in a physical controller.

Here is what the virtual thumbsticks look like on an iPhone:



For more information, see the [WWDC 2021 talk on game controllers](#).



This concludes our exploration of camera controllers and input mapping. Next time we will look at a notoriously hard problem — transparency — and some solutions for it.

# Day 24: Transparency



Warren Moore ·

7 min read · May 5, 2022

*This series of posts is my attempt to present the Metal graphics programming framework in small, bite-sized chunks for Swift app developers who haven't done GPU programming before.*

*If you want to work through this series in order, start [here](#). To download the sample code for this article, go [here](#).*

So far, we have only rendered opaque surfaces, objects that do not transmit any light through them. But many substances in the real world, from glass to water to gemstones, allow some light to pass through them, often tinting the light via selective absorption. We call such substances *transparent*.

You might be used to the term transparent referring to completely “clear” objects, while objects that allow some light through are called “translucent.” In this text, we will reserve the word “translucent” for substances that scatter light while transmitting it, and “transparent” will only be applied to substances that allow light through without seeming to scatter it. This latter property is sometimes called “optical transparency.”

As with all techniques in real-time graphics, we will use approximations to model light and materials in our treatment of transparency. Most particularly, in this article, all transparent surfaces will be treated as if they are infinitely thin. This implies that they will not exhibit scattering, nor selective absorption during transmission.

We will ultimately implement transparency in this article with a technique called *alpha blending*, which allows us to combine colors that are already in the frame buffer with partially transparent fragment colors, creating the

illusion of transparent objects.

Alpha blending can be understood in the context of the broader subject of *compositing*. First, we will look more closely at the alpha channel, then learn from the computer graphics literature how to use it.

## ***The Alpha Channel***

Many images and textures include a channel called the *alpha channel*, which we have mostly ignored up until this point. Alongside the red, green, and blue components of a color, the alpha component represents the color's *opacity*. By opacity, we mean the degree to which a surface prevents light from being transmitted. We call the inverse of this property the *transparency*, the degree to which a surface allows transmission.

There is another possible interpretation of the alpha channel. We can also think of alpha as representing *coverage*, the degree to which a surface “exists” at a given point. Imagine the mesh on a screen door: from a certain distance, such a surface restricts the transmission of light in almost the same way as tinted glass. The mesh selectively transmits light by blocking it entirely where the mesh is present, or allowing it to pass entirely where the mesh is absent. The aggregate effect is that the light is dimmed as if the door were partially transparent.

It turns out that both of these perspectives (the “opacity” perspective and the “coverage” perspective) are valid and useful at different times. In the remainder of this article, we will mostly use the opacity perspective.

## ***A Theory of Composition***

*Compositing* is the practice of combining colors or elements of two or more images. For example, when a weatherperson stands in front of a green screen so an animated map can be rendered behind them. By using the green portions of the camera frame as a mask, the compositing system selectively draws the map “behind” the presenter.

One early work on compositing in the context of computer graphics is



Porter and Duff's paper "Compositing Digital Images" (1984). The Porter-Duff composition model provides a mathematical framework for combining partially transparent elements together to achieve a variety of effects. The various ways of combining elements are called "operators."

Each operator acts on two colors, called "A" and "B", to produce a composited color. In this article we will only consider one of Porter and Duff's operators: the *A over B* operator. This operator has the effect of compositing A in front of B, making A the foreground and B the background.

When we implement alpha blending below, our background will be the existing contents of the frame buffer, and the foreground will be the current fragment. When the alpha component of the fragment is 0 (fully transparent), the background remains as-is. When the alpha component is 1 (opaque), the fragment completely replaces the background. Alpha values between 0 and 1 cause the colors to be blended together according to the foreground opacity.

## ***Premultiplication***

We say that an image is *premultiplied* when the RGB components of its pixels have been multiplied by the alpha component. For example, a premultiplied pixel that is semi-transparent red would have normalized components (0.5, 0, 0, 0.5). A nonpremultiplied semi-transparent red pixel would have components (1, 0, 0, 0.5).

Textures and render targets should contain premultiplied colors. The reason has to do with interpolation. Suppose you have two neighboring texels, one opaque red (1, 0, 0, 1) and one transparent black (0, 0, 0, 0). If the GPU samples the texture halfway between these two texels, linear interpolation produces the value (0.5, 0, 0, 0.5). If we treat this color as premultiplied, it represents a semi-transparent red. On the other hand, if we treat it as nonpremultiplied, we would weight it by its alpha during compositing, which would result in the final color being abnormally dark.



Now that we know about compositing and premultiplied alpha in the abstract, let's talk about how transparency is implemented by GPUs.

Alpha blending is the process of combining together the contents of the render buffer with the color returned by the fragment function. In most APIs, this blending is a *fixed-function* operation, meaning we configure it rather than writing a shader to do it.

Rather than using the foreground/background terminology from compositing, we will refer to the fragment color as the *source* and the render target color as the *destination*. This emphasizes that the fragment is being composited *into* the frame buffer by the alpha blending process.

Configuring a GPU for alpha blending consists of choosing blending *factors* and *operations*. Factors determine the respective weights of the inputs, while operations determine how the weighted contributions are combined. We might write the fully general blending operation as a set of two equations, one that determines the composited RGB components, and one that determines the composited alpha component.

$$\begin{aligned}C_{RGB} &= f_{sourceRGB} * A_{RGB} \oplus f_{destinationRGB} * B_{RGB} \\ C_{\alpha} &= f_{source\alpha} * A_{\alpha} \oplus f_{destination\alpha} * B_{\alpha}\end{aligned}$$

This is horribly abstract, though. As an example, let's consider how to implement Porter and Duff's A-over-B operator, which we might call "source-over-destination," or just "source-over" blending.

Our colors are assumed to be premultiplied, so we will not multiply the source by its alpha component — the source factor is simply one. Because we want the foreground to cover the background proportional to the alpha component of the fragment, the destination factor is one minus the source's alpha. We will use the same factors for the RGB and alpha channels and combine them with addition.

Our simplified equations for source-over blending are these:



$$C_{RGB} = A_{RGB} + (1 - A_{\alpha}) * B_{RGB}$$

$$C_{\alpha} = A_{\alpha} + (1 - A_{\alpha}) * B_{\alpha}$$

This looks just like ordinary linear interpolation, except that the source factor is one rather than the source alpha. This difference is accounted for by premultiplication.

## Alpha Blending in Metal

In Metal, alpha blending is achieved by configuring the render pipeline state for each color attachment.

We first enable blending for an attachment by setting its `isBlendingEnabled` property:

```
renderPipelineDescriptor.colorAttachments[0]
    .isBlendingEnabled = true
```

We then set the factors and operations for the source and destination terms of the RGB and alpha channels respectively. Here is how we configure premultiplied source-over blending:

```
renderPipelineDescriptor.colorAttachments[0]
    .sourceRGBBlendFactor = .one
renderPipelineDescriptor.colorAttachments[0]
    .destinationRGBBlendFactor = .oneMinusSourceAlpha
renderPipelineDescriptor.colorAttachments[0]
    .rgbBlendOperation = .add

renderPipelineDescriptor.colorAttachments[0]
    .sourceAlphaBlendFactor = .one
renderPipelineDescriptor.colorAttachments[0]
    .destinationAlphaBlendFactor = .oneMinusSourceAlpha
renderPipelineDescriptor.colorAttachments[0]
    .alphaBlendOperation = .add
```

You may recall that we have already been returning premultiplied colors

from our fragment shaders, in anticipation of using alpha blending:

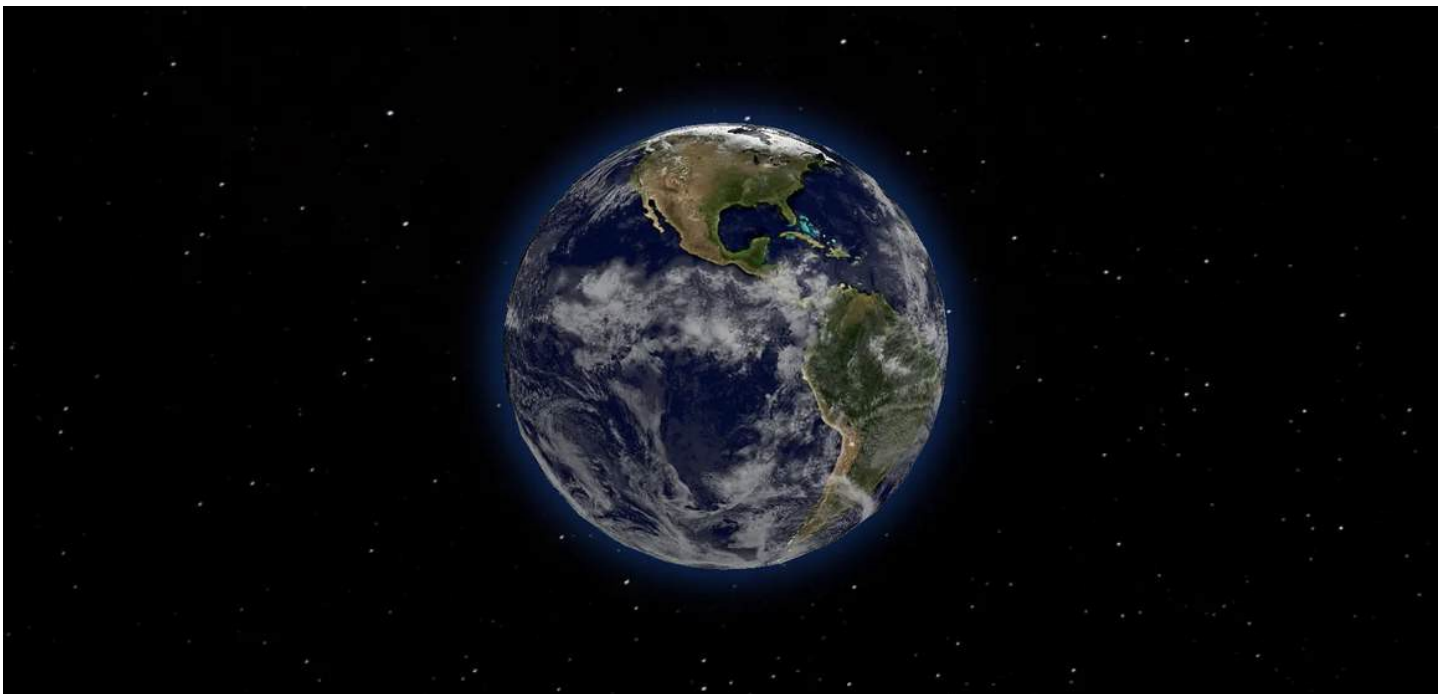
```
return float4(litColor * baseColor.a, baseColor.a);
```

Enabling blending on the render pipeline state and returning transparent colors from the fragment function are sufficient to achieve alpha blended results.

Our sample scene consists of a model of Earth in space. The opaque planetary surface is surrounded by a transparent layer of clouds, and a faint, transparent glow wraps the planet to emphasize the atmosphere's depth.

We load the planet scene from three separate OBJ files so we can move the parts independently. Each frame, we rotate the earth and cloud spheres to suggest the passage of time, while also orienting the plane containing the atmospheric glow so it always faces the camera.

Running the sample app gives us a view of home from a distance:



## ***Order Dependency in Blending***

Not all is right in the world, however. There is a dirty little secret in transparency that makes getting correct blended results much harder. The inconvenient truth is that *order matters* when compositing. If we combine

multiple layers of transparent materials together in an arbitrary order, we

will not get a correct result.

Mathematically, we say that the blending equations are not *commutative*: in general, operation  $\oplus$  followed by operation  $\otimes$  is not the same as operation  $\otimes$  followed by operation  $\oplus$ . This implies that we need to enforce an order for our transparent surfaces. In most cases, we want to render surfaces from far to near (i.e., from back to front).

Sorting surfaces from back to front works in many cases. However, sorting transparent surfaces suffers from the same issues we saw when contemplating the painter’s algorithm and the z-buffer: it is not always possible to sort triangles unambiguously by depth. Even when it is possible, sorting a dynamic scene every frame can be prohibitively expensive.

There have been many attempts over the years to implement “order-independent transparency” (OIT), but every possible approach has downsides. We handily avoided ordering issues in our sample scene by manually determining the order of the transparent elements, but this is not always possible in dynamic scenes. If you are interested in various mitigations for order dependency, I recommend consulting the bibliography of [McGuire and Bavoil’s 2013 paper](#).

Next time, we will consider how to add higher fidelity to our transparent objects with environment-mapped reflection and refraction.

# Day 25: Environment Mapping



Warren Moore ·

9 min read · May 10, 2022

*This series of posts is my attempt to present the Metal graphics programming framework in small, bite-sized chunks for Swift app developers who haven't done GPU programming before.*

*If you want to work through this series in order, start [here](#). To download the sample code for this article, go [here](#).*

So far, we have only created two-dimensional textures, but Metal has other texture types. Textures can be three-dimensional; these are sometimes called *volume textures*. Textures can also be *array textures*, which contain a number of elements, where each element is a 2D texture and each element has the same width and height. Finally, textures can be *cube textures*, which have six faces arranged in a cube shape. These faces are referred to as *slices*.

Each of these three texture types — 3D, 2D array, and cube — have three dimensions in a certain sense. What differs among them is how they are used. 3D textures can be used to store volumetric data, such as density. 2D array textures can store sequences of animated images, such as the frames of a GIF. Cube textures are commonly used to store images of the environment surrounding a scene.

In this article, we will take a deeper look at cube textures, and particularly how to cheaply implement reflection and refraction using environment maps.

## Cube Maps

Cube textures are colloquially called *cube maps*. A cube map has six faces



arranged in a cube around the origin. Although a cube map consists of six separate images, they are almost always designed to connect seamlessly at the edges, which is what allows them to store a 360-degree view of an environment.

There are numerous ways of storing cube map contents on disk. They can be stored as six discrete image files, or combined into a “strip” image, with the faces laid out sequentially in either a horizontal or vertical direction. One such “vertical strip” image is shown below.





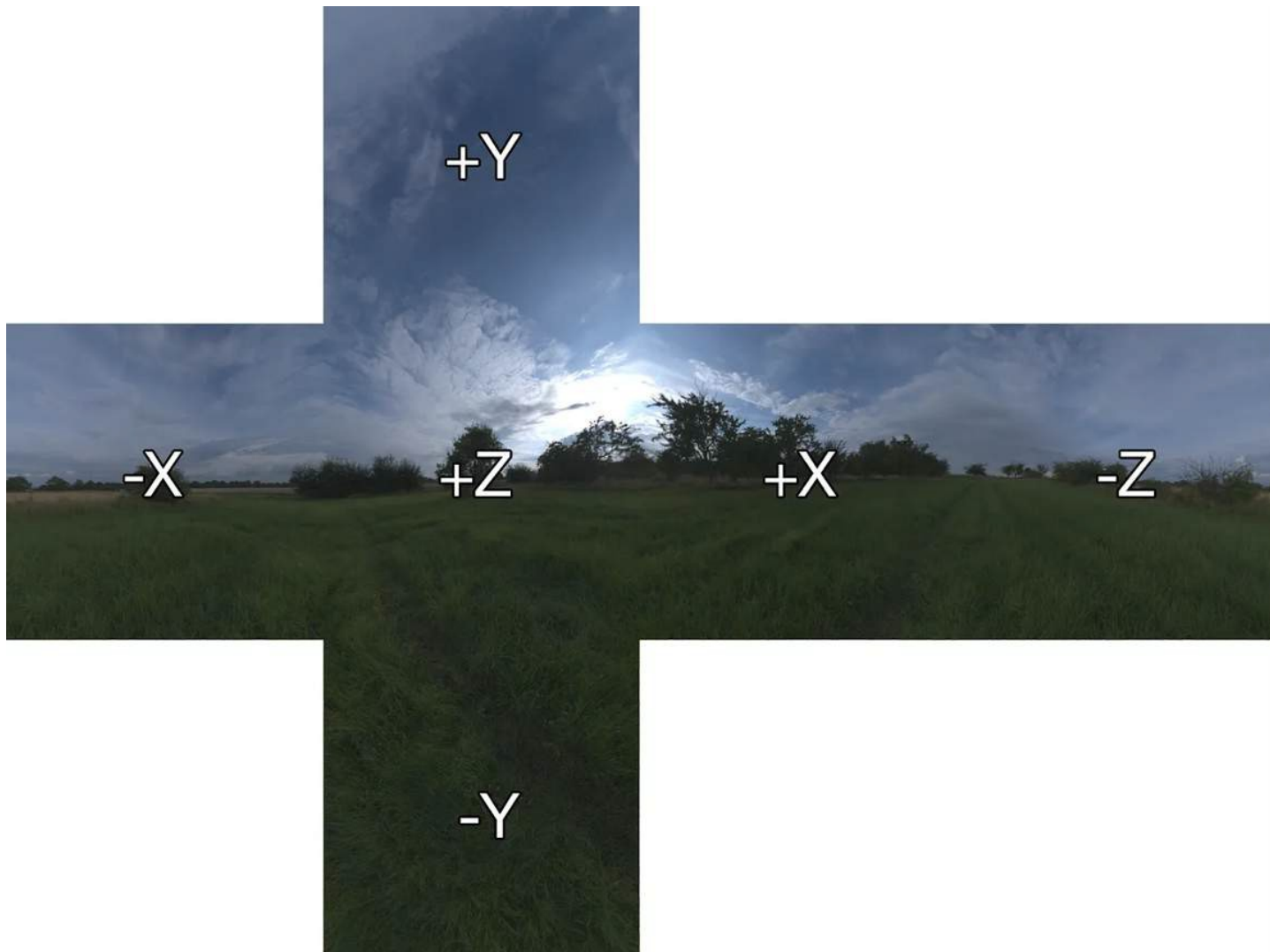








Cube maps are also sometimes displayed in a “cross” configuration, which better illustrates the adjacency of the faces:



Storing a cube map in this way tends to waste space; it is shown here for illustrative purposes.

To create a cube texture in Metal, we first create and populate a texture descriptor:

```
let cubeSize = 256
let textureDescriptor = MTLTextureDescriptor()
textureDescriptor.textureType = .typeCube
textureDescriptor.pixelFormat = .bgra8Unorm
textureDescriptor.width = cubeSize
textureDescriptor.height = cubeSize
textureDescriptor.mipmapLevelCount = mipmapLevelCount(for: cubeSize)
```

Here, we have described a cube texture with a pixel format of `MTLPixelFormat.bgra8Unorm` and a size of 256x256 with mipmaps down to 1x1 (9 levels).

`MTLTextureDescriptor` also has a static method for filling out a cube texture descriptor. Here is an equivalent definition to the one above:



```
MTLTextureDescriptor.textureCubeDescriptor( pixelFo
    rmat: .bgra8Unorm,
    size: cubeSize,
    mipmapped: true)
```

In either case, we are responsible for setting the storage mode and usage flags that correspond to our intended use:

```
textureDescriptor.usage = .shaderRead
textureDescriptor.storageMode = .private
```

Regardless of how the cube image is stored, it requires a little work to load the faces into a Metal cube texture.

First, we need a mapping from the face orientation to the face index. This is given in the table below.

Face Index	Face Orientation
0	+X
1	-X
2	+Y
3	-Y
4	+Z
5	-Z

To copy into all six faces of the texture, we loop over the faces (slices) and call the

```
replace(region:mipmapLevel:slice:withBytes:bytesPerRow:bytesPerImage:)
```

method on the texture, providing the face’s image data:

```
let bytesPerRow = cubeSize * 4
let bytesPerImage = bytesPerRow * cubeSize
let region = MTLRegionMake2D(0, 0, cubeSize, cubeSize)
for faceIndex in 0..<6 {
    let imageBytes = /* load image data */
    cubeTexture.replace(region: region,
```





```

        mipmapLevel: 0,
        slice: faceIndex,
        withBytes: imageBytes,
        bytesPerRow: bytesPerRow,
        bytesPerImage: bytesPerImage)
    }

```

If this seems tedious, don't worry. Fortunately, `MTKTextureLoader` understands how to create and populate cube textures on our behalf.

`MTKTextureLoader` only supports one cube image layout:

`MTKTextureLoader.CubeLayout.verticalStrip`. Including the `.cubeLayout`

option in our texture loader options prompts MetalKit to treat the image as a vertical strip image and make a cube texture to store it. The vertical strip image's height must be *exactly* six times its width.

If you want the texture loader to also generate mipmaps on your behalf, include the `.generateMipmaps` option.

```

let cubeTextureOptions: [MTKTextureLoader.Option : Any] = [
    .textureUsage : MTLTextureUsage.shaderRead.rawValue,
    .textureStorageMode : MTLStorageMode.private.rawValue,
    .generateMipmaps : true,
    .cubeLayout : MTKTextureLoader.CubeLayout.vertical
]

```

For here, we can use any of the `newTexture...` APIs to load our cube map:

```

let texture = try?
    textureLoader.newTexture( URL: textureURL,
    options: cubeTextureOptions)

```

With our cube texture loaded and ready to go, it's time to talk about how we use cube maps in our virtual scenes.

## Environment Maps

*Environment mapping* is a technique used to approximate reflection and refraction effects by using a texture to store an image of a scene and its

surroundings. Environment maps can be *static* or *dynamic*. Static environment maps are often loaded from image files, while dynamic environment maps are rendered from the scene contents itself, potentially using a static environment map to fill in distant detail.

Environment mapping encompasses a variety of effects, including environment-mapped backgrounds, reflections, and refractions. We will discuss each of these in turn in the sections below.

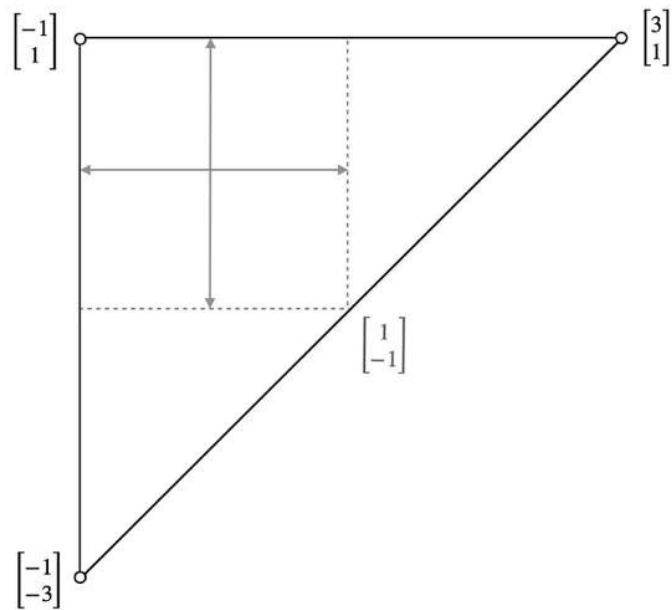
## ***Environment-Mapped Background***

To provide additional realism and coherence to a rendered scene, the environment map is sometimes rendered as the scene background. This can be done by constructing a *skybox*, a cubical geometry that surrounds the camera and is textured with the environment cubemap. Other geometry types are possible: paraboloid domes and hemispheres are sometimes used. In any case, the vertices of the sky geometry are ordered such that the surface normals point inward, toward the viewer.

There are some downsides to using a mesh that surrounds the scene to render the background. For one, the background mesh's position must be synchronized to the camera position so that it remains at a constant distance from the viewer. For another, distant geometry requires special handling so that it doesn't get culled by the camera's far plane. Finally, even if the geometry is carefully chosen, about half of it will be outside the view frustum on average.

Instead of using a skybox mesh or other geometry, we will render a single *full-screen triangle* to draw the background. How can we cover a rectangular viewport with a single triangle? By thinking outside the box ("the box" being clip space, in this case).

In clip space, the viewport spans from -1 to 1 on the X and Y axes. So, by situating vertices as shown in the figure below, we cover every pixel in the viewport.



A vertex function that renders a full-screen triangle is shown below.

```
struct FullscreenVertexOut
{
    float4 position
    [[position]]; float4
    clipPosition;
};

vertex FullscreenVertexOut
vertex_fullscreen_env( uint index [[vertex_id]])
{
    float2 positions[] = {
        { -1, 1 },
        { -1, -3 },
        { 3, 1 }
    };
    FullscreenVertexOut out;
    out.position = float4(positions[index], 1, 1);
    out.clipPosition = out.position;
    return out;
}
```

There are a couple of things to note here.

First, the output structure has a `position` member and a `clipPosition` member, both of which are assigned the same value: the clip-space position. Why the redundancy? Recall that the position we receive in the fragment function is in *viewport coordinates*; it's had the perspective divide and viewport transformation applied to it. On the other hand, if we pass the clip-space position separately, it is interpolated just like any other varying value, and we can easily return to world space by applying the inverse view-

projection matrix.

The other thing to note is that we force the z and w values of the clip-space position to 1, which pushes the vertex out to the maximum view distance. This is useful since we want the sky to appear behind anything else that has been drawn.

We can now write a fragment function that transforms the clip-space triangle positions into world space, turns the position vector into a direction vector, and samples the environment map to draw the background:

```
fragment float4 fragment_fullscreen_env(
    FullscreenVertexOut in [[stage_in]],
    constant float4x4 &inverseViewProjectionMatrix [[buffer(0)]],
    texturecube<float, access::sample> environmentTexture
[[texture(0)]],
    sampler cubeSampler [[sampler(0)]])
{
    float4 worldPosition = inverseViewProjectionMatrix *
        in.clipPosition;
    float3 worldDirection =
        normalize( worldPosition.xyz);
    return environmentTexture.sample(cubeSampler, worldDirection);
}
```

Note that the texture coordinates we use to sample the cube map are three-dimensional: they represent a direction pointing from the center of the cube map. The cube map is sampled at the position where this direction vector intersects the cube. This distinguishes cube map sampling from all other kinds of texture sampling in Metal.

To draw the background, we encode a single-triangle draw call. The code to bind the appropriate resources and encode the draw can be extracted into a method called `drawEnvironment(_:)`:

```
func drawEnvironment(
    _ renderCommandEncoder: MTLRenderCommandEncoder)
{
    renderCommandEncoder.setRenderPipelineState( environmentRenderPipelineState)

    renderCommandEncoder.setFragmentBytes(
        &clipToViewDirectionTransform,
        length: MemoryLayout<float4x4>.size,
        index: 0)

    renderCommandEncoder.setFragmentTexture(
```



```
environmentTexture, index: 0)
renderCommandEncoder.setFragmentSamplerState(
    samplerState, index: 0)

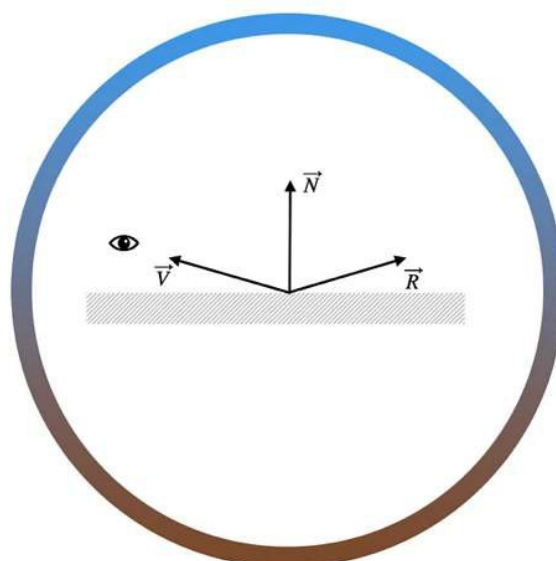
renderCommandEncoder.drawPrimitives(
    type: .triangle, vertexStart: 0, vertexCount: 3)
}
```

We use the `setFragmentBytes(_:length:index:)` method to pass in a previously-constructed matrix that concatenates the inverse projection matrix and the inverse of the rotational part of the view matrix. This matrix takes a clip-space position to a world-space direction, which is what we want to sample our environment texture with.

## Reflection

Environment maps can also be used to cheaply simulate reflection. Without an environment map, we would have to use more sophisticated techniques such as raytracing to calculate how light reflects from shiny surfaces. With an environment map, simple reflections can be obtained with a single texture sample.

We have already considered the physics of reflection, so we won't revisit all of that math again, but here is a schematic diagram of what we are doing when we implement environment-mapped reflection. The circular gradient represents the environment map surrounding the scene; think of it as a longitudinal slice through a sky sphere:





Essentially, we reflect the view direction vector around the surface normal to get a reflection vector. This reflection vector then becomes our 3D environment map texture coordinate.

The Metal Shading Language has a utility function called `reflect` that does the necessary reflection math for us. We just need to make sure that the inputs (the vector from the camera to the surface and the surface normal) are in a compatible coordinate space. Since we already need the vectors `v` and `N` to perform lighting, we use these to find the reflection vector. Then we transform the resulting vector from view space to world space.

The fragment shader snippet for implementing environment-mapped reflections is shown below.

```
float3 reflection = reflect(-V, N);  
float3 R = frame.inverseViewDirectionMatrix * reflection;  
float3 envColor = environmentTexture.sample(cubeSampler, R).rgb;
```

Once we have the environment color, we can use it as a radiance value for computing an additional lighting term alongside our previous direction and point lights.

For illustration purposes, here is a fully-reflective torus rendered with environment-mapped reflection:

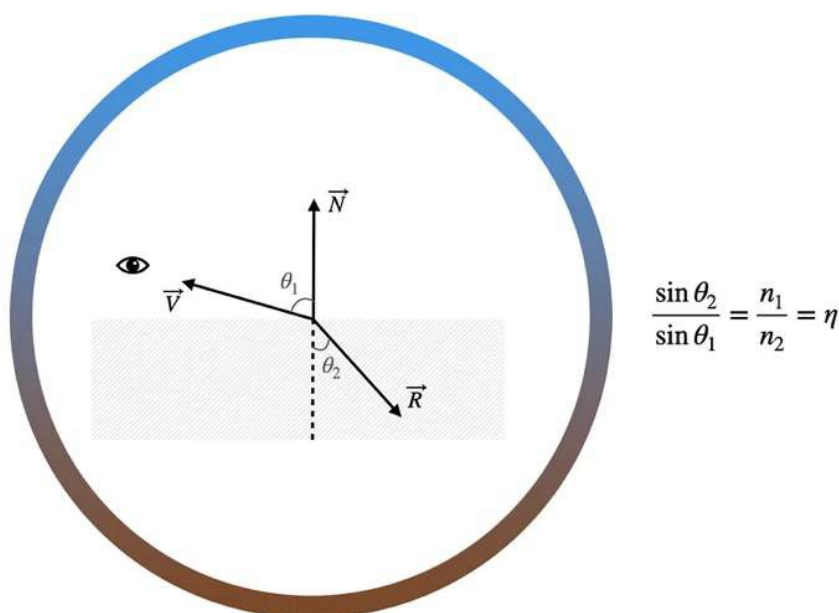




## Refraction

Another effect we can achieve cheaply with environment mapping is *refraction*. Refraction makes light rays appear to bend as they travel cross a boundary (“interface”) between a material with one index of refraction and a material with a different index of refraction. For example, an interface between air (with an index of refraction very near 1) and water (with an index of refraction of about 1.33). The apparent bending occurs because of the difference in the speed of light through the two media.

To implement refraction, we need to know the respective index of refraction of each of the materials we are modeling. Specifically we need their ratio. This ratio, called *eta*, is equal to the inverse ratio of the sines of the angles between the incident vector and the refracted vector. This relationship is illustrated below.



As with reflection, MSL has a utility function called `refract` that computes the refracted vector, given the incident vector, the normal, and the ratio `eta`.

We have the option of hardcoding our indices of refraction or supplying them as material parameters to the fragment shader. In the snippet below, we assume the existence of constants that hold the IOR of air and water, and find the ratio by division.

```
float eta = iorAir / iorWater;  
float3 refraction = refract(-V, N, eta);  
float3 R = frame.inverseViewDirectionMatrix * refraction;  
float3 envColor = environmentTexture.sample(cubeSampler, R).rgb;
```

The image below shows the effect of refraction using an environment map.



Although there are more sophisticated ways to render reflective and refractive materials, it doesn't get much cheaper than environment mapping.

Continuing with our theme of special effects and texture mapping, next time we will look at normal mapping.

# Day 26: Normal Mapping



Warren Moore ·

8 min read · May 12, 2022

*This series of posts is my attempt to present the Metal graphics programming framework in small, bite-sized chunks for Swift app developers who haven't done GPU programming before.*

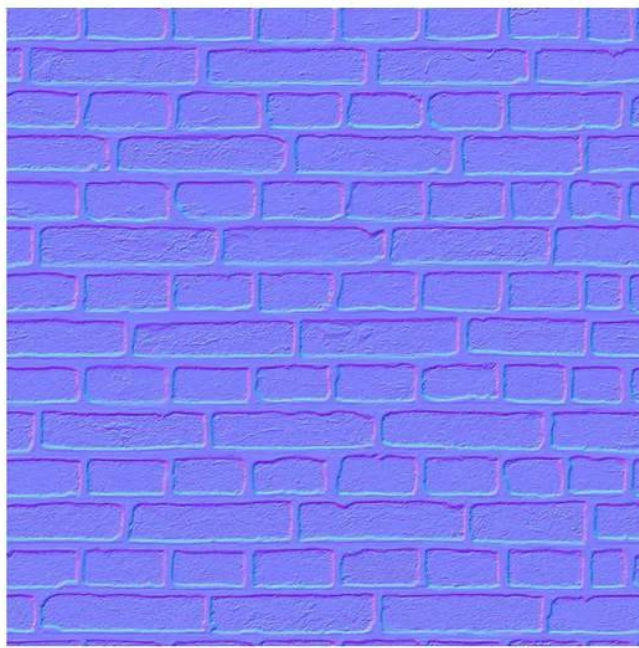
*If you want to work through this series in order, start [here](#). To download the sample code for this article, go [here](#).*

Last time, we looked at how to use environment mapping to cheaply add a modicum of realism to reflective and refractive materials. This time, we consider how to create the illusion of finer surface detail without adding extra geometry, using a technique called normal mapping.

## **Normal Mapping**

*Normal mapping* is a rendering technique that supplies additional apparent surface detail in the form of a texture map. In the same way that we add color information in between vertices with ordinary texture mapping, we add variation in the surface normal with a *normal map*.

We have been using interpolated vertex normals along the way to compute per-pixel lighting. Normal mapping recomputes the normal on a per-pixel basis according to the normals encoded in a normal map. So what does a normal map look like?



The most common normal map type has a purplish hue. Normals are perpendicular to the surface, meaning they are predominantly aligned with the local Z axis. As a vector, the local +Z direction is the vector  $[0 \ 0 \ 1]$ .

Interpreted as a color with normalized components, this is pure blue. But the normals in a normal map need to be transformed so that negative direction components can be encoded as positive color components. This is done by adding 1 to the normal's components and dividing the result by 2.

The resulting vector can represent any unit vector by using components between 0 and 1. The purplish tinge of normal maps comes from the fact that the +Z direction is encoded to the color vector  $[0.5 \ 0.5 \ 1]$ , which is a light purple. Normals that point in the +X direction have a red tint, while normals that point in the +Y direction have a green tint, as you might expect.

## ***Tangent Space***

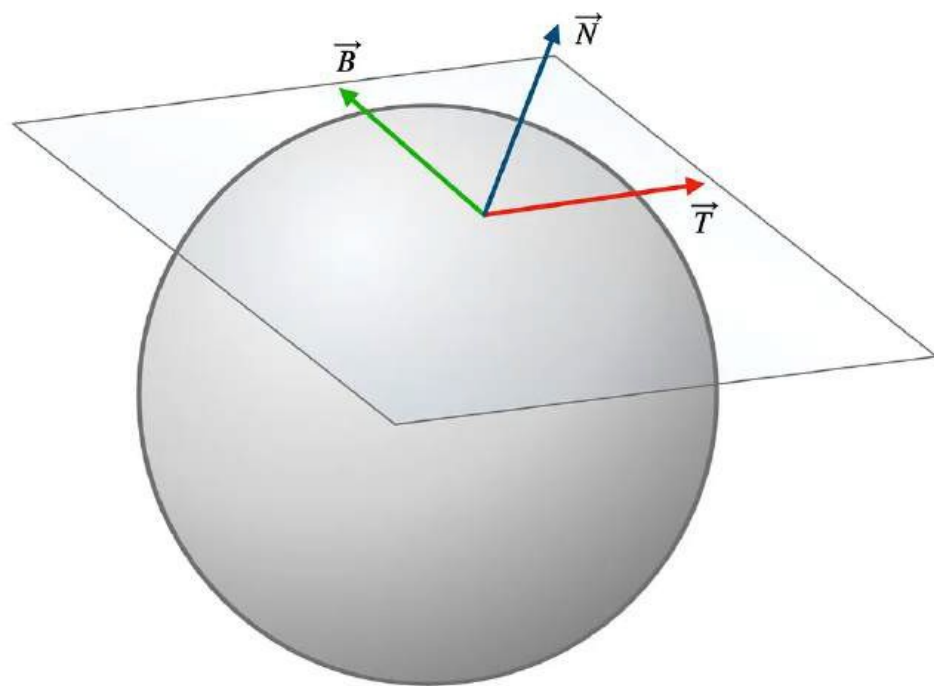
To understand how to interpret these encoded normals and work with them in our shaders, we need to understand which coordinate system they are in. This is a new coordinate space called *tangent space*.

Unlike model space and world space, tangent space can be different at every point on a model's surface. Specifically, tangent space is an orthonormal vector basis whose Z axis is aligned with the surface normal, and whose X and Y axes lie in the tangent plane of the surface. The tangent-space X axis is called the *tangent*, and the Y axis is called the *bitangent*.



Just knowing that tangent space is an orthonormal basis doesn't help use chose its X and Y axes, however: there are infinitely many orthonormal bases for any choice of the Z axis. So we need additional information to determine the tangent and bitangent directions.

Fortunately, we have additional information in the form of texture coordinates. At a given point, the u and v coordinates change at a particular rate that depends on the texture mapping parameterization we chose when texture-mapping our mesh. We can use these *texture-space derivatives* to disambiguate tangent space, by choosing the tangent to point in the direction of most rapid change in the u coordinate, and likewise with the bitangent and the v coordinate.



The complete math for this is out of the scope of this article, but we will discuss how to go about generating normal maps in the next section.

## ***Normal Map Generation***

To generate a normal map, we start with a highly tessellated version of the source mesh, called a “high-poly” mesh. The version we will render at run- time has far fewer triangles; this is our “low-poly” mesh. The normal map essentially stores the difference in surface detail between the high-poly and low-poly mesh.

The process of normal map generation consists of constructing a tangent space basis at each texel and encoding the transformation between the coarse-grained interpolated vertex normal and the fine-grained surface normal. Most modern 3D modelers (like Maya and Blender) include features for producing normal maps from high-poly meshes; this procedure is called “normal map baking.”

It is important that the procedure used to generate a normal map is the exact inverse of the procedure used to apply it during lighting. Disparities between conventions and processing can produce unsightly artifacts, as detailed in [this Ben Golus article](#). Fortunately, as time goes on, the [MikkTSpace](#) convention is becoming dominant across the visual effects and game communities.

## ***Generating Tangents with Model I/O***

We need per-vertex tangents to reconstruct tangent space in our shaders. Many model formats support storing vertex tangents along with vertex positions and normals. Sometimes, though, tangents are missing, either because the format doesn’t support them or because they were not exported.

We can use Model I/O to construct tangents automatically from the other data in a mesh (namely the normals and texture coordinates). After loading a mesh, we call the

`addTangentBasis(forTextureCoordinateAttributeNamed:normalAttributeNamed:tangentAttributeNamed:)` method on `MDLMesh`. This method uses the existing data in the texture coordinate and normal attributes to populate the tangent attribute.

```
mdlMesh.addTangentBasis( forTextureCoord
    inateAttributeNamed:
MDLVertexAttributeTextureCoordinate,
    normalAttributeNamed: MDLVertexAttributeNormal,
    tangentAttributeNamed: MDLVertexAttributeTangent)
```

After this method, the mesh is not guaranteed to conform to the vertex



descriptor which was supplied when it was loaded. We can fix this by creating a new Model I/O vertex descriptor that includes a tangent attribute laid out as we prefer:

```
vertexDescriptor.vertexAttributes[0].name =
    MDLVertexAttributePosition
vertexDescriptor.vertexAttributes[0].format = .float3
vertexDescriptor.vertexAttributes[0].offset = 0
vertexDescriptor.vertexAttributes[0].bufferIndex = 0

vertexDescriptor.vertexAttributes[1].name =
    MDLVertexAttributeNormal
vertexDescriptor.vertexAttributes[1].format = .float3
vertexDescriptor.vertexAttributes[1].offset = 12
vertexDescriptor.vertexAttributes[1].bufferIndex = 0

vertexDescriptor.vertexAttributes[2].name =
    MDLVertexAttributeTangent
vertexDescriptor.vertexAttributes[2].format = .float4
vertexDescriptor.vertexAttributes[2].offset = 24
vertexDescriptor.vertexAttributes[2].bufferIndex = 0

vertexDescriptor.vertexAttributes[3].name =
    MDLVertexAttributeTextureCoordinate
vertexDescriptor.vertexAttributes[3].format = .float2
vertexDescriptor.vertexAttributes[3].offset = 40
vertexDescriptor.vertexAttributes[3].bufferIndex = 0

vertexDescriptor.bufferLayouts[0].stride = 48
```

We then re-set the mesh's vertex descriptor, causing it to lay out the vertex data as expected:

```
mdlMesh.vertexDescriptor = vertexDescriptor
```

We also change the input type for our vertex function to include the tangent:

```
struct VertexIn {
    float3 position  [[attribute(0)]];
    float3 normal    [[attribute(1)]];
    float4 tangent   [[attribute(2)]];
    float2 texCoords [[attribute(3)]];
};
```

Note that `tangent` is a four-element vector rather than a three-element vector like `position` and `normal`. This is because the fourth element of the tangent encodes the *handedness* of the tangent basis: a value of `1` indicates that it is right-handed, while a value of `-1` indicates that it is left-handed. We use this to correctly reconstruct tangent space in our shaders.

## ***Tangent Space and Texture Space***

Speaking of handedness, you may have noticed that the tangent space illustrated above is right-handed, while Metal’s texture space is left-handed (with `v` increasing downwards). How do we account for this?

Separate from all other tangent space conventions, we have to consider the texture space convention of our API. Normal maps are commonly said to use the “OpenGL convention,” which uses a lower-left origin and upward- increasing V axis, or the “DirectX convention,” which uses an upper-left origin and downward-increasing V axis. This difference primarily manifests as OpenGL-style normal maps seeming to be lit “from above” with green highlights, while DirectX-style maps seem lit “from below.”

Metal’s convention matches DirectX, so we either need to use normal maps that already agree with this convention, or invert the green channel when sampling. The sample code expects Direct-X style normal maps.

## ***Implementing Normal Mapping in Metal***

We will continue to perform lighting in view space, since that allows us to offload work to the CPU and vertex shader while retaining maximum precision.

To construct tangent space in the fragment shader, we need to pass the view-space normal and tangent out of our vertex function. We update the output structure as follows:

```
struct VertexOut {
    float4 position [[position]];
    float3 viewPosition;
    float3 normal;
    float3 tangent;
```

```
float tangentSign [[flat]];
float2 texCoords;

};
```

Note that we also carry the tangent sign into the fragment shader so we can flip the tangent basis if required.

We also slightly update the `InstanceConstants` structure to take a 3x3 matrix that stores the inverse transpose of the model-view matrix. (This is the correct matrix to use to properly transform vectors into view space; we’ve been neglecting it so far, since it frequently has the same effect as transforming by the model-view matrix itself.)

```
struct InstanceConstants {
    float4x4 modelMatrix;
    float3x3 normalMatrix;
};
```

The body of the vertex function proceeds in the ordinary way, with the addition of the view-space tangent computation:

```
float4 worldPosition = instance.modelMatrix *
    float4(in.position, 1.0);
float4 viewPosition = frame.viewMatrix * worldPosition;

VertexOut out;
out.position = frame.projectionMatrix * viewPosition;
out.viewPosition = viewPosition.xyz;
out.normal = normalize(instance.normalMatrix * in.normal);
out.tangent = normalize(instance.normalMatrix * in.tangent.xyz);
out.tangentSign = in.tangent.w;
out.texCoords = in.texCoords;
```

In the fragment shader, our first goal is to construct a rotation matrix that transforms from tangent space to view space. This is commonly called a “TBN” matrix, since its columns are the tangent, bitangent, and normal, respectively.

```
float3 T = normalize(in.tangent);
float3 B = cross(in.normal, in.tangent) * in.tangentSign;
```



```
float3 Nv = normalize(in.normal);  
float3x3 TBN = { T, B, Nv };
```

Note that instead of passing in the bitangent, we reconstruct it as the cross product of the normal and tangent (flipping it with the tangent's sign if needed). `Nv` is the normalized view-space normal, which is what we'd use for lighting calculations if we weren't normal-mapping.

To decode the tangent-space normal, we first sample the normal map, then apply the inverse of the encoding calculation:

```
float3 Nt = normalTexture.sample(linearSampler, in.texCoords).xyz;  
Nt = Nt * 2.0 - 1.0;
```

To get our lighting normal, we transform the tangent-space normal into view space:

```
float3 N = TBN * Nt;
```

The remainder of our lighting calculations remain unchanged.

The sample app includes a scene with a couple of different normal-mapped materials. The figure below illustrates the effect of normal mapping, which looks even more convincing in motion.





The illusion produced by normal mapping is far from perfect. When viewed in silhouette and at oblique angles, the low-poly nature of the normal-mapped mesh becomes apparent. Other techniques like parallax mapping and displacement mapping can alleviate this somewhat, with each approach carrying its own computational cost.

Next time we will look at tessellation, a method for generating additional geometry on the fly. Tessellation can be used in concert with other techniques like displacement mapping to add geometric detail without a proportionate increase in vertex attribute memory.

# Day 27: Tessellation



Warren Moore ·

12 min read · Jun 2, 2022

*This series of posts is my attempt to present the Metal graphics programming framework in small, bite-sized chunks for Swift app developers who haven't done GPU programming before.*

*If you want to work through this series in order, start [here](#). To download the sample code for this article, go [here](#).*

Last time, we looked at how to add the illusion of finer surface detail by using a normal map to vary the lighting normal at a higher rate than the underlying vertex normals.

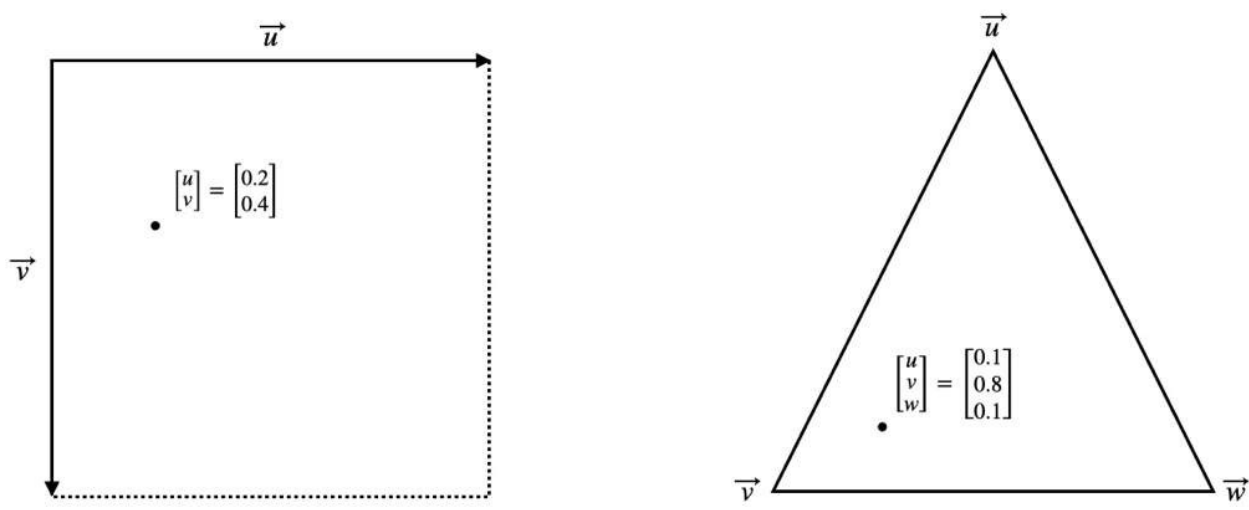
This time, we will look at tessellation, a technique for generating more geometry dynamically without needing to allocate more vertex storage. Tessellation works by subdividing the primitives in a mesh in a systematic way, producing additional, smaller triangles.

To perform tessellation, we will issue draw calls for a new kind of primitive: patches. A *patch* is a quadrilateral (“quad”) or triangular coordinate space combined with a set of control points, which influence how the subdivided geometry is positioned. We will call the patch coordinate space a “domain.”

Quad and triangle domains are parameterized differently. Positions in a quad patch have two parameters, commonly labeled  $u$  and  $v$ , similar to how we denote texture coordinates. Triangle domains have three coordinates, labeled  $u$ ,  $v$ , and  $w$ . These are barycentric coordinates, meaning they express how much influence each vertex of the domain has on the position. Points inside such a patch have coordinates that sum to 1.



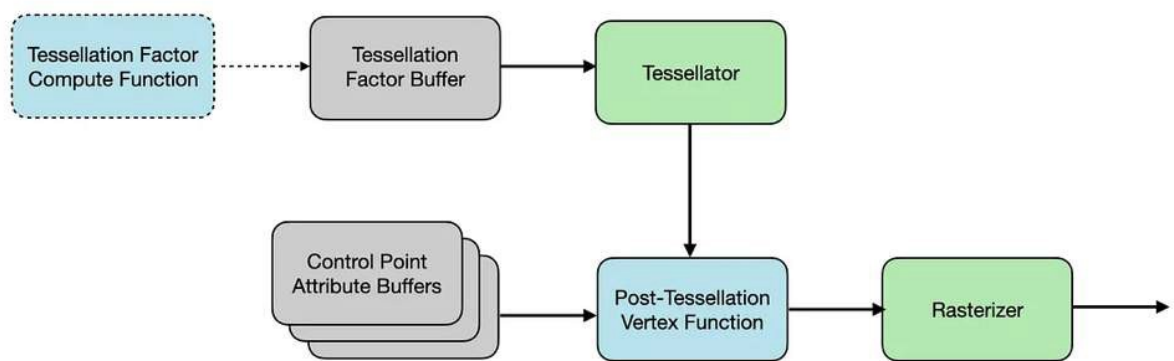
These two different coordinate schemes are illustrated below with example positions.



Rather than vertices, patches have *control points*. Like vertices, control points have associated attributes. Unlike vertices, multiple control points are fetched at once in the vertex function. We are responsible for writing a function that interprets the control point data according to our chosen subdivision scheme. Furthermore, the number of control points is not necessarily equal to the number of vertices commonly associated with the patch domain. Patches can have between 0 and 32 control points.

Patches differ from ordinary mesh geometry in that they are not processed by the regular vertex pipeline. Instead, they are subdivided before the vertex shader runs, according to a fixed-function stage called the *tessellator*. The tessellator operates by fetching the tessellation factors for a patch, subdividing the patch, and passing the resulting vertices to the vertex shader.

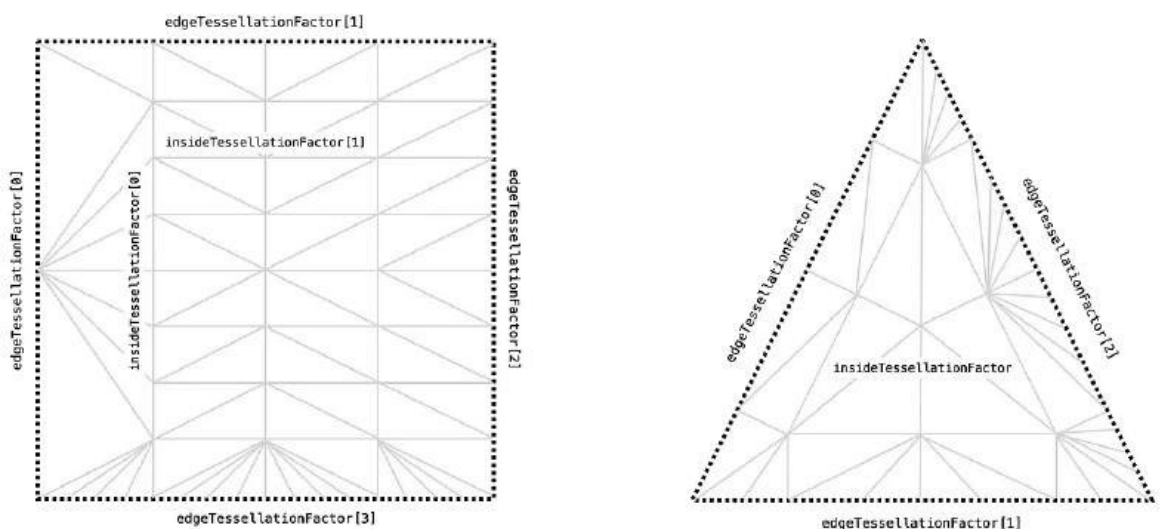
The modified graphics pipeline is illustrated below. An optional compute function may be used to populate the tessellation factor buffer. The tessellation factors are consumed by the fixed-function tessellator, which subdivides the patch domain accordingly. Then, the subdivided geometry passes to the post-tessellation vertex stage. We will discuss all of these steps in greater detail below.



## Tessellation Factors

Recall from our previous discussion of primitives that Metal cannot render quadrilaterals; they must be triangulated first. Since tessellation is a fixed- function stage, we have limited control over how the subdivision of patches occurs. The two most important inputs to the tessellator are the tessellation factors and the partition mode.

Tessellation factors are numbers that determine how many times each region of the tessellation domain should be subdivided. A quad domain has a total of six factors: four edge factors and two inside factors. Edge factors determine how many times each edge should be subdivided, while inside factors determine how many times the interior region should be subdivided. Triangular tessellation factors are similar, except there are three edge factors (corresponding to the three edges of the triangle) and only one inside factor, corresponding to the interior region. These various factors are illustrated below.



Why two different kinds of factors? One reason is that if adjacent patches

have different edge factors, or if the vertex function produces different positions for vertices that are supposed to be coincident, cracks will appear. If we can control the subdivision of the interior of the patch separately from its edges, we can potentially use a lower factor inside and whatever factor is needed to avoid cracking along the edges, allowing us to allocate vertices where they are needed most.

The partition mode determines subtler aspects of how the subdivided vertices are distributed. This is discussed below in the implementation section.

## ***Authoring Meshes for Tessellation***

A 3D artist must choose which patch type to use when designing a mesh. Different subdivision types are suitable for different uses. Quad patches are commonly used with subdivision schemes like Catmull-Clark subdivision surfaces. Triangle patches can be used with other schemes like Loop subdivision and PN triangles.

Generating additional geometry is not very useful by itself: lighting and texturing are commonly done per-pixel, so without the addition of other information, more vertices just mean more work for the GPU. The power of tessellation comes from the fact that we can supply additional information in the form of textures and control point attributes. Most 3D modeling packages support exporting various per-control point attributes such as normals, tangents, and texture coordinates.

It is important to maintain the topology of the model through all stages from authoring to rendering. Asset import libraries sometimes prefer to triangulate input meshes so that they are compatible with real-time renderers, but triangulating a quad patch destroys the information needed to render the mesh correctly with tessellation. We will see below how to use Model I/O to import patch geometry without disrupting the authored topology.

## ***A Patch Mesh Class***

Since we will be drawing patches rather than triangles, we need a slightly different data model for our submeshes. Specifically, rather than a primitive type, our submesh will have a patch type and a control point count. It will also have a patch count derived from the index count and control point count.

The `PatchSubmesh` interface looks like this:

```
enum PatchType {
    case tri
    case quad
}

class PatchSubmesh {
    let patchType: PatchType
    let patchControlPointCount: Int
    let patchCount: Int
    let indexBuffer: MeshBuffer
    let indexType: MTLIndexType
    let indexCount: Int
    var material: Material?
```

Using this interface, we can easily draw each patch with a draw call.

Now that we know how to author and represent patch meshes, let's look at how to actually use tessellation in Metal.

## ***Tessellation in Metal***

To perform tessellation in Metal, we have three tasks:

1. Write tessellation factors to a tessellation factor buffer.
2. Issue patch primitive draw calls.
3. Write a post-tessellation vertex function that transforms tessellated vertex positions and produces other interpolated vertex attributes.

## ***Tessellation Factors***

We can accomplish task #1 in a variety of ways depending on what we're trying to do. In the simplest case, we can use a constant set of tessellation

factors that apply to every patch. In other scenarios, we might want to take other information, such as the patch's distance from the camera, into account to calculate tessellation factors for each patch. Updating factors can also be done at any rate we choose: we might write our tessellation factors once and reuse them across frames, or we might generate new tessellation factors every frame.

In the interest of keeping as much work as possible on the GPU, it is common to use compute functions to populate the tessellation factor buffer. Compute functions have the virtue of having direct access to GPU memory, and most algorithms we might use to determine tessellation factors can be implemented in a compute kernel for efficient parallel computation.

Tessellation factors in Metal are half-precision floating-point numbers; each factor occupies two bytes. Each patch type has a different tessellation factor structure, corresponding to the tessellation factor diagram above.

The declarations of these structures in MSL are shown below.

```
struct MTLQuadTessellationFactorsHalf
{
    half edgeTessellationFactor[4];
    half insideTessellationFactor[2];
};

struct MTLTriangleTessellationFactorsHalf
{
    half edgeTessellationFactor[3];
    half insideTessellationFactor;
};
```

The interpretation of these factors is dependent on the configuration of the render pipeline state, specifically the *tessellation partition mode*.

Here is a simplified definition of Metal's `MTLTessellationPartitionMode` enumeration:

```
enum MTLTessellationPartitionMode {
    case pow2
    case integer
    case fractionalOdd
    case fractionalEven
}
```

The `.pow2` mode is the most restrictive. The tessellation factors are rounded up to the nearest power of two, and the subdivided positions are evenly distributed along the relevant axis. In the `.integer` mode, the tessellation factors are rounded up to the nearest integer. For information on the other two partition modes, consult [Apple's documentation](#).

In the sample code, we somewhat arbitrarily choose the `.integer` partition mode:

```
renderPipelineDescriptor.tessellationPartitionMode = .integer
```

We also don't bother computing tessellation factors dynamically; instead they are written into the constant buffer alongside the rest of the constant data.

## ***Patch Draw Calls***

To issue patch draw calls, we use one of the draw call methods on the render command encoder, just as we do when drawing ordinary meshes. There are indexed and non-indexed variants of these methods. We will focus here on the

```
drawIndexedPatches(numberOfPatchControlPoints:patchStart:patchCount:patchIndexBuffer:patchIndexBufferOffset:controlPointIndexBuffer:controlPointIndexBufferOffset:instanceCount:baseInstance:) method. This method takes a lot of parameters, so let's break it down.
```

The number of patch control points is the number of control points per patch. Recall that this is not necessarily the number of vertices in the corresponding primitive (3 for triangles and 4 for quads). Instead, each patch can have as many control points as are required to implement the desired subdivision scheme. By coincidence, the simple subdivision scheme we use in the sample code uses four control points per quad patch.

The `patchStart` and `patchCount` parameters determine the base patch index and number of patches to draw.

The `patchIndexBuffer` and its corresponding offset are used to specify a patch index buffer, which indicates the indices of the patches to render. This is distinct from the control point index buffer, which holds indices into the control point attribute buffers (much like an ordinary index buffer).

These parameters are optional and we will set them to `nil` and `0` respectively.

The `controlPointIndexBuffer` and its corresponding offset specify the control point index buffer.

We will not perform instanced rendering, but the final two parameters allow us to specify the instance count and base instance index.

As with ordinary primitive rendering, we bind the mesh's attribute buffers before issuing any draw calls. When tessellating, we additionally bind the tessellation factor buffer on the render command encoder:

```
renderCommandEncoder.setTessellationFactorBuffer( constantBuffer  
    r,  
    offset: tessellationFactorOffset,  
    instanceStride: 0)
```

(As mentioned above, for simplicity's sake we just use the constant buffer to store the tessellation factors. In a real application this might be a dedicated buffer, and the factors might be computed dynamically.)

Using our `PatchSubmesh` class from above, we can now issue a patch draw call as follows:

```
let indexBuffer = submesh.indexBuffer  
renderCommandEncoder.drawIndexedPatches(  
    numberOfPatchControlPoints: submesh.patchControlPointCount,  
    patchStart: 0,  
    patchCount: submesh.patchCount,  
    patchIndexBuffer: nil,  
    patchIndexBufferOffset: 0,  
    controlPointIndexBuffer: indexBuffer.buffer,  
    controlPointIndexBufferOffset: indexBuffer.offset,  
    instanceCount: 1,  
    baseInstance: 0)
```





## The Post-Tessellation Vertex Function

After the tessellator runs, we are responsible for figuring out where each vertex in the domain should be positioned in clip space, as well as calculating any other vertex attribute positions we might want interpolated, such as normals, tangents, and texture coordinates.

In this context, the vertex function is called a *post-tessellation vertex function*, because it runs after the tessellator. The vertex function operates on control points rather than on input vertices.

A post-tessellation vertex function must be preceded by a `patch` attribute that specifies the patch type and (optionally on iOS) the number of patch control points. It also takes a `float2` parameter with the `[[position_in_patch]]` that Metal fills with the coordinates of the vertex in the patch domain. Here is a simplified function signature:

```
[[patch(quad, 4)]]
vertex VertexOut vertex_quad(
    patch_control_point<VertexIn> controlPoints [[stage_in]],
    float2 positionInPatch [[position_in_patch]])
```

The `patch_control_point` template takes a type parameter that defines the attributes of the control points. This type depends on what kind of data we need to generate interpolated vertices. In the simplest case, it can just be the same struct that we use to define our vertex attributes:

```
struct VertexIn {
    float3 position [[attribute(0)]];
    float3 normal [[attribute(1)]];
    float4 tangent [[attribute(2)]];
    float2 texCoords [[attribute(3)]];
};
```

Since we are using the `attribute` attribute (confusing, I know), these values will be fetched in the same way that vertex data is ordinarily fetched when a



vertex descriptor is provided during render pipeline state construction. However, we have to update our vertex descriptors to accommodate this. In particular, we need to set the step function to

```
MTLVertexStepFunction.perPatchControlPoint rather than the default  
MTLVertexStepFunction.perVertex :
```

```
vertexDescriptor.layouts[0].stepFunction = .perPatchControlPoint
```

With this change, Metal will fetch control point data from our vertex buffers and populate the `controlPoints` parameter with all of the control point data for the surrounding patch each time the vertex function is called.

Notably, Model I/O does not have a distinct notion of patch primitives, so a `MTLVertexDescriptor` translated from an `MDLVertexDescriptor` needs to be fixed up by setting the step function when tessellation is being used.

Inside the vertex function itself, we are responsible for calculating whatever vertex properties we want from the control points. This is where we implement our chosen subdivision scheme.

A very helpful utility function for such calculations is `bilerp`, which performs bilinear interpolation among arbitrary values of a common type, such as `float4`. Bilerp can be implemented as follows:

```
template <typename T>  
T bilerp(T c00, T c01, T c10, T c11, float2 uv) {  
    T c0 = mix(c00, c01, T(uv[0]));  
    T c1 = mix(c10, c11, T(uv[0]));  
    return mix(c0, c1, T(uv[1]));  
}
```

To use the `bilerp` function in the vertex function, we supply it with the control point attribute data we want to interpolate. In the simple case of interpolating just the position, the vertex function would look something like this:

```
[[patch(quad, 4)]]
vertex VertexOut
    vertex_simple_quad( patch_control_point<VertexIn>
        controlPoints [[stage_in]], float2 positionInPatch
        [[position_in_patch]])
    {
        float3 p00 = controlPoints[0].position;
        float3 p01 = controlPoints[1].position;
        float3 p10 = controlPoints[3].position;
        float3 p11 = controlPoints[2].position;
        float3 position = bilerp(p00, p01, p10, p11, positionInPatch);

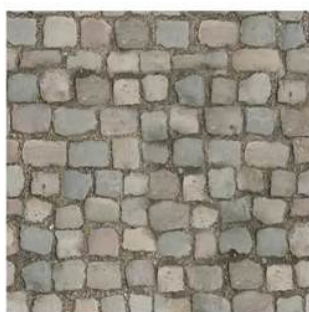
        // ...
    }
}
```

We read the position of each control point and pass them all into `bilerp`, along with the `positionInPatch` parameter, which tells us where in the patch we are. The `bilerp` function returns the model-space position of the interpolated vertex, which we can then process and transform in the ordinary ways.

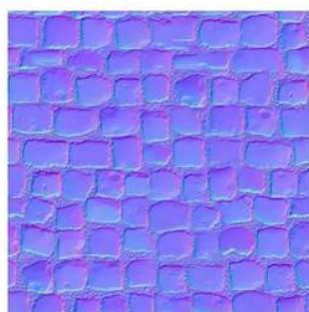
## Displacement Mapping

Now that we know how to write post-tessellation vertex functions, let's put all of that subdivided geometry to work. We will implement a technique called *displacement mapping*, which uses a heightmap-like texture to displace, or move, vertices along the surface normal. This can produce intricate geometry that affects a shape's silhouette in a way that naive normal mapping cannot.

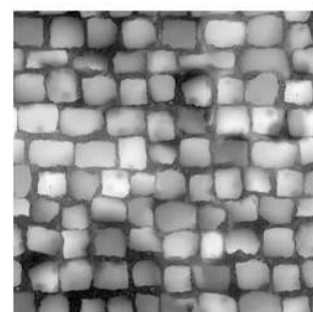
To implement displacement mapping, we first need a displacement map. It will complement our existing base color and normal maps:



Base Color



Tangent-Space Normals



Displacement

We first expand our `Material` type with an optional displacement texture

member:

```
class Material {
    var baseColor = SIMD4<Float>(1, 1, 1, 1)
    var baseColorTexture: MTLTexture?
    var normalTexture: MTLTexture?
    var displacementTexture: MTLTexture?
}
```

Model I/O looks for an attribute named `map_disp` when loading MTL material files that accompany OBJ models, so we can use this directive to specify the displacement map:

```
map_Kd cobblestone_baseColor.png
map_tangentSpaceNormal cobblestone_normal.png
map_disp cobblestone_displacement.png
```

and later populate the displacement texture property during model loading:

```
if let displacementProperty = mdlMaterial.property(
    with: MDLMaterialSemantic.displacement) {
    if displacementProperty.type == .texture {
        if let textureURL = displacementProperty.urlValue
            { material.displacementTexture =
                try? textureLoader.newTexture(URL: textureURL,
                                                options: textureOptions)
            }
    }
}
```

The post-tessellation vertex function needs the displacement map and the surface normal to perform displacement mapping, so we use our `bilerp` utility function to find the normal and other vertex properties:

```
[[patch(quad, 4)]]
vertex VertexOut
    vertex_displace_quad( patch_control_point<VertexIn>
        controlPoints [[stage_in]], constant InstanceConstants
        *instances [[buffer(2)]], constant FrameConstants &frame
        [[buffer(3)]],
        texture2d<float, access::sample> displacementMap [[texture(0)]],
        float2 positionInPatch [[position_in_patch]],
```



```
uint instanceID [[instance_id]])
{
    constant InstanceConstants &instance =
        instances[instanceID];

    float3 p00 = controlPoints[0].position;
    float3 p01 = controlPoints[1].position;
    float3 p10 = controlPoints[3].position;
    float3 p11 = controlPoints[2].position;
    float3 position = bilerp(
        p00, p01, p10, p11, positionInPatch);

    float3 n00 = controlPoints[0].normal;
    float3 n01 = controlPoints[1].normal;
    float3 n10 = controlPoints[3].normal;
    float3 n11 = controlPoints[2].normal;
    float3 normal = bilerp(
        n00, n01, n10, n11, positionInPatch);

    // ... other attributes ...
}
```

To determine the degree of displacement for a given vertex, we sample the displacement map at the (interpolated) texture coordinates and take just the red channel

```
constexpr sampler bilinearSampler(coord::normalized,
                                   filter::linear,
                                   mip_filter::none,
                                   address::repeat);

float displacement =
    displacementMap.sample( bilinearSampler,
                           texCoords).r;
```

To determine the position of the displaced vertex, we combine the interpolated original position, the sampled displacement, and the displacement factor (which we can vary interactively for demonstration purposes, or bake into the asset).

```
position += normal * displacement * frame.displacementFactor;
```

Below is an exaggerated example of displacement mapping, showing how highly subdivided geometry, combined with displacement mapping and normal mapping, produces a more realistic appearance than normal mapping alone.



With this, we have achieved our goal: implementing tessellation and displacement mapping in Metal. Next time, we will look at how to bring meshes to life with vertex skinning.



# Day 28: Skinning



Warren Moore ·

14 min read · Jul 30, 2022

*This series of posts is my attempt to present the Metal graphics programming framework in small, bite-sized chunks for Swift app developers who haven't done GPU programming before.*

*If you want to work through this series in order, start [here](#). To download the sample code for this article, go [here](#).*

We have seen some simple examples of time-based animation of object transformations, but this time, we will look at the powerful combination of animation and hierarchy.

The purpose of the Model I/O framework is to provide a unified interface to a variety of 3D asset formats. For this reason, it provides abstractions for a number of common concepts, such as meshes, materials, and animations. We have already gotten acquainted with how to load meshes and materials with Model I/O and render them with Metal.

Another foundational concept implemented in Model I/O is the *skeleton*. A skeleton consists of a set of joints, which are arranged in a hierarchy, just like the node hierarchies we have already worked with. By associating mesh vertices with one or more joints, we can realistically animate the mesh just by animating the transformations of the joints.

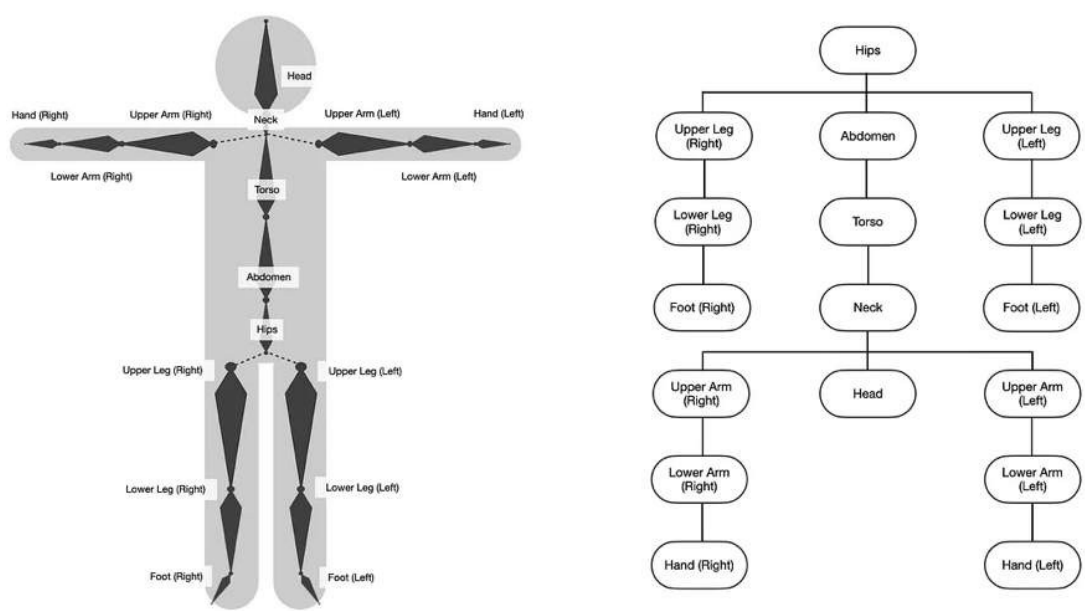
In this article, we will discuss how to load a model containing a skeleton and an animation that changes the transforms of the skeleton's joints over time, a process called skeletal animation.

# Rigging and Skinning

To perform skeletal animation, we first need a model that has been appropriately rigged and skinned.

*Rigging* is the process of creating a skeleton for the mesh. During rigging, the artist selects the set of joints that will allow them the desired amount of control over the movement of the various parts of the model. Many 3D modeling applications include pre-built “rigs” that match common scenarios like bipedal humanoids or quadrupedal creatures.

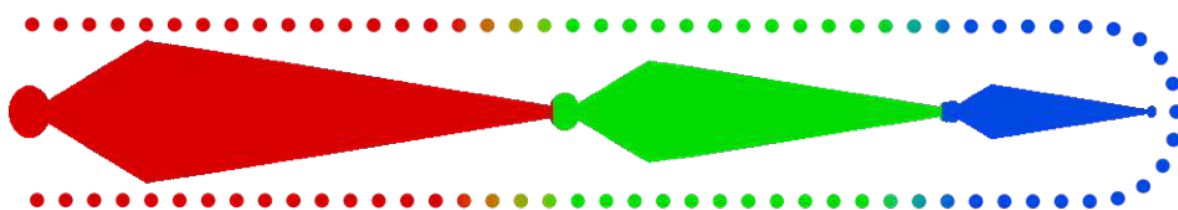
The figure below provides two different views of the same skeleton. On the left side, the bones’ physical arrangement is shown, overlaid on a bipedal figure. On the right side, the joints are arranged hierarchically, which emphasizes the fact that the “Hips” joint is the root of the hierarchy.



Joints often correspond to common body parts like feet, hips, and spines, but they don’t have to. Skeletons are usually simplified relative to the real- world objects they represent, both for performance reasons and to make animation easier. On the other hand, features like ponytails which do not contain bones in the real world might be rigged with joints in a skeleton to allow artistic control over hair movement during animation. The general rule is that if part of a model needs to be moved independently, it should have one or more associated joints.

After rigging, the artist performs a task called *skinning*. During skinning, the artist associates each vertex in the mesh with one or more joints. Each joint — vertex pair has a *weight* that determines the joint’s influence over the vertex. When the joint is animated, the vertex moves in proportion to this weight.

The figure below is a representation of joint-vertex associations. Each dot is a vertex, with the dot’s color illustrating the joint or joints it is influenced by. For example, a fully red dot is only influenced by the upper arm joint (shown in red), while dots closer to the “elbow” are more yellowish in hue to show that they are also partially influenced by the lower arm joint.



The purpose of allowing more than one joint to influence a vertex is so that vertices can move more naturally. If each vertex moved relative to a single joint, the overall motion would appear rigid and not at all lifelike. With the weights allocated as shown above, the arm can flex and bend much more realistically. Done well, skinning can enable much more expressive and flexible animation than the rigid body transforms we used previously.

## ***Skeletal Animation***

The kinds of animations we have seen so far have applied to a single object’s transformation. In order to fully unlock the power of animation, we now need to understand *hierarchical animation*. When we say hierarchical animation, we mean motion of a node relative to its ancestors in a transformation hierarchy, rather than in world space.

One kind of hierarchical animation we can achieve with a rigged mesh is *skeletal animation*. The hierarchy of joints in a mesh's skeleton gives us the ability to express relative position and orientation of different parts of a mesh. By animating these positions and orientations over time, we can create graceful articulated figures.

Now that we understand the fundamentals of rigging, skinning, and joint hierarchies, let's look at how to implement skeletal animation in Metal. We will begin with how skeletons are represented in Model I/O.

## ***Skeletons in Model I/O***

Skeletons in Model I/O are represented by the `MDLSkeleton` class. This type does not contain `MDLObject`s representing the joints, as you might expect. Instead, it contains a list of *joint paths*, which are slash-separated strings of names. This list defines the skeletal hierarchy very compactly, in a way that is reminiscent of nested file paths.

For the skeleton in the figures above, here is a portion of the joint paths list:

```
/Hips
/Hips/Abdomen
/Hips/Abdomen/Torso
/Hips/UpperLegLeft
/Hips/UpperLegLeft/LowerLegLeft
...
```

Here is a simplified declaration for the `MDLSkeleton` class:

```
class MDLSkeleton : MDLObject
{
    var jointPaths: [String]
    var jointBindTransforms: MDLMatrix4x4Array
    var jointRestTransforms: MDLMatrix4x4Array
    public init(name: String, jointPaths: [String])
}
}
```

Along with the joint paths, the skeleton has two lists of transforms: the bind transforms and the rest transforms.



A *bind transform* moves a joint from its local coordinate space into the coordinate space of its parent joint. When every joint has its bind transform applied, the resulting arrangement is called the *bind pose* or reference pose.

From the bind pose, an animator creates an animation by repositioning the joints in a sequence of desired poses, which are stored as a sequence of keyframes. Each keyframe stores a snapshot of the transformation of each animated node at the corresponding key time. At runtime these transformations are interpolated over time to play back the animation.

In contrast, a *rest transform* is the transform applied to a joint that is not being animated. Since some animations do not affect every joint, the rest transform is the transform applied by default after the bind transform.

The `MDLMatrix4x4Array` type abstracts access to an array of 4x4 matrices. We can request matrices whose elements are `float`s or `double`s from such an object. Since we use `float`s in our shaders, we will use the `float4x4Array` property of these objects below, which is typed as `[float4x4]`, a good match for shader code.

To keep track of the transformation of each joint, we will need a data type that holds a local transformation (which might be animated) and can provide a concatenated matrix representing the joint-space-to-model-space transformation.

## ***A Skeleton Class***

It turns out we already have a type that holds a transform and can concatenate transforms: the *Node* class. Although we don't need all of its features, we may as well reuse it rather than creating a special type just for joints. We will create a hierarchy of nodes for each skeleton in the Model I/O asset, instantiating one node per joint (I will refer to such nodes as “joint nodes” below).

The joint nodes will not belong to the scene's node hierarchy. Instead, we need a separate object to hold and manage the joint nodes. This will be our `Skeleton` class. In addition to holding the joint nodes themselves, the

skeleton will hold the bind transforms and the rest transforms of each joint. The data

members of the `Skeleton` class look like this:

```
let name: String
let jointPaths: [String]
let inverseBindTransforms: [float4x4]
let restTransforms: [float4x4]

var jointCount: Int
    { return joints.count
  }

private var joints = [Node]()
```

We copy the list of joint paths from the `MDLSkeleton` so we can match up joint nodes to their respective paths during animation.

To construct a `Skeleton` from an `MDLSkeleton`, we first copy the name, joint paths, bind transforms, and rest transforms. In this process, we invert the bind transforms. Then, we generate the joint node hierarchy and assign each joint its resting transform.

```
init(_ mdlSkeleton: MDLSkeleton) {
    name = mdlSkeleton.name
    jointPaths = mdlSkeleton.jointPaths
    inverseBindTransforms =
mdlSkeleton.jointBindTransforms.float4x4Array.map { $0.inverse }
    restTransforms = mdlSkeleton.jointRestTransforms.float4x4Array
    joints = makeSkeletonHierarchy(from: jointPaths)

    for (jointIndex, joint) in zip(0..., joints)
        { joint.transform = restTransforms[jointIndex]
    }
}
```

The `makeSkeletonHierarchy` method is responsible for building the joint node hierarchy. We can do this with a two-pass algorithm. In the first pass, we instantiate nodes and build a dictionary that maps from joint paths to joint nodes:

```
func makeSkeletonHierarchy(from jointPaths: [String]) -> [Node] {
```

```

var joints = [Node]()
var jointsForPaths = [String : Node]()
for jointPath in jointPaths {
    let joint = Node()
    joint.name = jointPath
    jointsForPaths[jointPath] = joint
    joints.append(joint)
}

```

In the second pass, we build the hierarchy by iterating over the nodes, finding the parent of each node, and adding the node to its parent. We also reset each node’s name to its “unqualified” name rather than its full path. For example, a joint with the path

Root/Body/Hips/Abdomen/Torso/Neck/Head/Ear1\_R would wind up with the name Ear1\_R.

```

for jointPath in jointPaths {
    let child = jointsForPaths[jointPath]!
    let parentPath = (jointPath as
NSString).deletingLastPathComponent as String
    let parent = jointsForPaths[parentPath]
    child.name = (jointPath as NSString).lastPathComponent as String
    parent?.addChildNode(child)
}

return joints
}

```

With our skeleton assembled, let’s look at how to render a skinned mesh with Metal.

## ***Vertex Skinning in Metal***

Like many of the techniques we have implemented, vertex skinning requires relatively little shader code. Most of the complexity arises from keeping track of the various transformation matrices and preparing them for use within the vertex function. In this section, we will first look at how to gather the (potentially animated) transformations of the joints and then consider how to use them to skin our vertices in the vertex function.

Just as we have been doing with our other constant data, we will write the joint transformation matrices into a buffer so they can be accessed in the vertex function. We do this by iterating over the joint nodes in the skeleton





and asking each node for its world transform, then writing the result into the constant buffer.

To apply the joint transforms in the vertex function, we need two additional pieces of data for each vertex: the joint indices and the joint weights. The joint indices tell us *which* joints should influence the vertex, and the weights tell us *how much* influence each joint has.

Since we need this data on a per-vertex basis, we add two vertex attributes — called `jointWeights` and `jointIndices` — to our vertex descriptor.

`jointIndices` is a `ushort4`, a four-element vector of unsigned 16-bit integers, and `jointWeights` is a `float4`. This implies that up to four joints can influence each vertex, which is normally a good balance between performance and expressiveness.

The expanded vertex structure in our shader code looks like this:

```
struct SkinnedVertexIn {  
    [[attribute(0)]];  
    [[attribute(1)]];  
    float2 texCoords [[attribute(2)]];  
    ushort4 jointIndices [[attribute(3)]];  
    float4 jointWeights [[attribute(4)]];  
};
```

We send our joint transforms into the vertex function by adding a new buffer parameter of type `constant float4x4 *`:

```
vertex VertexOut  
skinned_vertex_main( SkinnedVertexIn  
in [[stage_in]],  
...  
constant float4x4 *jointMatrices [[buffer(3)]],  
...)  
{
```

Vertex skinning affects the model-space position of the vertex, so it is applied *before* the model matrix. To determine the “skinning matrix” that takes us from model space to skinned model space, we index into the joint

transform array using the current vertex's joint indices, then add the joint matrices together, weighted by the current vertex's joint weights:

```
float4 modelPosition = float4(in.position, 1.0);
float4 modelNormal = float4(in.normal, 0.0);

float4x4 skinningMatrix =
    in.jointWeights[0] * jointMatrices[in.jointIndices[0]] +
    in.jointWeights[1] * jointMatrices[in.jointIndices[1]] +
    in.jointWeights[2] * jointMatrices[in.jointIndices[2]] +
    in.jointWeights[3] * jointMatrices[in.jointIndices[3]];

modelPosition = skinningMatrix * modelPosition;
modelNormal = skinningMatrix * modelNormal;
```

We get the skinned vertex position by multiplying the skinning matrix by the vertex's model-space position, just as we apply any other transformation. The rest of the vertex function proceeds as normal.

There are no changes needed in the fragment function.

Since we now might be drawing a mix of objects that are skinned and objects that are not skinned, we need to create two render pipeline states (with different vertex descriptors and different vertex functions). We can then select between them at render time.

At this point, we have written all of the code necessary to load a skeleton and apply joint transformations to achieve vertex skinning. Here is the sample app showing a skinned, static 3D model:



A skinned mesh isn't much use without animation, though, so let's look at how to load and apply animations.

## ***Skeletal Animation in Model I/O***

Model I/O represents skeleton animations with the `MDLPackedJointAnimation` type. A joint animation consists of timed arrays of transform components (translations, rotations, and scales). These components can be sampled over time and composed to determine the animated transformations of the targeted joints. The “packed” part of the name indicates that the data for each value of each component are stored adjacently in memory (in contrast to sparse or strided storage).

Here is the interface for the `MDLPackedJointAnimation` class (simplified):

```
class MDLPackedJointAnimation : MDLObject, MDLJointAnimation {
    var jointPaths: [String]
    var translations: MDLAnimatedVector3Array
    var rotations: MDLAnimatedQuaternionArray
    var scales: MDLAnimatedVector3Array
}
```

The `jointPaths` property contains the names of the joints affected by the animation; not all joints in a skeleton need to be affected by every animation. The remaining properties are animated arrays containing transformation data; these arrays can be used to determine the translation, rotation, and scale of a joint at a particular moment in time.

We will write a simple wrapper type that makes working with Model I/O joint animations a little easier: the `JointAnimation` class. This class will hold the animation data, provide animation timing information, and compute lists of joint transformations for us.

Since Model I/O joint animations do not provide useful information like the start time or duration of the animation, we can add a couple of extensions that make this easier:

```
extension MDLPackedJointAnimation {
  var minimumTime: TimeInterval {
    return [translations, rotations, scales]
      .reduce(TimeInterval.greatestFiniteMagnitude) { return min($0,
$1.minimumTime) }
  }

  var maximumTime: TimeInterval {
    return [translations, rotations, scales]
      .reduce(-TimeInterval.greatestFiniteMagnitude) { return
max($0, $1.maximumTime) }
  }
}
```

We now begin to specify our own `JointAnimation` class, beginning with the public properties:

```
class JointAnimation {
  let name: String
  let jointPaths: [String]
  let startTime: TimeInterval
  let duration: TimeInterval
  let translations: MDLAnimatedVector3Array
  let rotations: MDLAnimatedQuaternionArray
  let scales: MDLAnimatedVector3Array
  //...
}
```

Initializing an animation just consists of copying the relevant properties, using the extensions above for the timing details:

```
init(_ animation: MDLPackedJointAnimation)
{ name = animation.name
  jointPaths = animation.jointPaths
  translations = animation.translations
```



```

rotations = animation.rotations
scales = animation.scales

startTime = animation.minimumTime
duration = animation.maximumTime - startTime
}

```

To produce the set of composed, animated transforms for a skeleton at a particular time, we implement the `jointTransforms(at:)` method:

```

func jointTransforms(at time: TimeInterval) -> [float4x4] {
    let translationsAtTime = translations.float3Array(atTime: time)
    let rotationsAtTime = rotations.floatQuaternionArray(atTime: time)
    let scalesAtTime = scales.float3Array(atTime: time)
    return zip(translationsAtTime, zip(rotationsAtTime,
scalesAtTime)).map {
        let (translation, (orientation, scale)) = $0
        return float4x4(translation: translation, orientation:
orientation, scale: scale)
    }
}

```

This completes the `JointAnimation` class definition.

Animations don't do anything on their own; they have to be *bound* to a node. Model I/O uses the `MDLAnimationBindComponent` component to associate animations with nodes.

We can find the animation bind components associated with a Model I/O object during loading. We will write a small utility extension to retrieve the animation binding if present:

```

extension MDLObject {
    var animationBind: MDLAnimationBindComponent?
    { return components.filter({
        $0 is MDLAnimationBindComponent
    }).first as? MDLAnimationBindComponent
    }
}

```

What is inside an animation bind component? Here is a simplified look at the `MDLAnimationBindComponent` class:





```
class MDLAnimationBindComponent : NSObject, MDLComponent {
    var jointAnimation: MDLJointAnimation?
    var skeleton: MDLSkeleton?
    var geometryBindTransform: matrix_double4x4
}
```

An animation bind component holds a reference to a joint animation (which is a protocol to which `MDLPackedJointAnimation` conforms). It also holds a reference to a skeleton, which allows us to match up the joint names in the animation with their corresponding joint nodes. Finally, it holds a “geometry bind transform”, which is a matrix that transforms skinned vertices into joint space. It is often the identity matrix—in fact, we assume it always is—but we mention it here for completeness.

When loading a Model I/O asset, we check each object for an animation binding and maintain a list of animations, so we can apply them later. We also ensure each animated node has a reference to its skeleton.

```
if let animationBinding = mdlObject.animationBind {
    if let mdlAnimation = animationBinding.jointAnimation as?
MDLPackedJointAnimation {
        let animation = JointAnimation(mdlAnimation)
        animations.append((animation, node))
    }

    if let mdlSkeleton = animationBinding.skeleton
    { node.skinner = Skinner(
        skeletonForMDLSkeleton(mdlSkeleton),
        float4x4(animationBinding.geometryBindTransform))
    }
}
```

## ***Animation Playback***

To play back an animation, we first need a notion of time. We keep track of the current time in a member variable and increment it by the frame duration each time we draw.

Once we have a global timeline established, we need to convert it into the “local time” of the animation. To find the local time, we first subtract the animation’s start time from the global time. The local time is the remainder of this result divided by the animation’s duration. Using the remainder

causes the animation to loop as time progresses.

We can add a small method to our `Node` class to start playing an animation:

```
func runAnimation(_ animation: JointAnimation)
    { self.animation = animation
  }
```

We also add a method to apply the current animation, if any, to the node's skeleton:

```
func update(at time: TimeInterval) {
  if let animation = animation, let skinner = skinner {
    let localTime = max(0, time - animation.startTime)
    let loopTime = fmod(localTime, animation.duration)
    skinner.skeleton.apply(animation: animation, at: loopTime)
  }
}
```

Here, we first compute the animation's local time, then tell our skinner's skeleton to apply the animation at that time.

To apply a skeletal animation, we supply the time to the animation's

`jointTransforms(at:)` method, which returns the current transforms of all animated joint nodes.

We then need to apply the animated transforms to the animated joint nodes and the rest transforms to the non-animated joint nodes. Here is the complete implementation of `Skeleton`'s `apply(animation:at:)` method:

```
func apply(animation: JointAnimation, at time: TimeInterval)
    { let animatedTransforms = animation.jointTransforms(at:
      time) for (skeletonJointIndex, jointPath) in zip(0...,
      jointPaths) {
      if let animationJointIndex = animation.jointPaths.firstIndex(of:
      jointPath) {
        joints[skeletonJointIndex].transform =
        animatedTransforms[animationJointIndex]
      } else
      { joints[skeletonJointIndex].transform
        =
```



```
}  
}
```

With these changes made, we can tell a node to play an animation and see our skinned animation system in action:



This is great, but playing a running animation isn't very useful if the character can't actually run around.

## ***Finding Nodes***

We will sometimes find it convenient to locate a node by name. We can do this by writing a recursive method on the `Node` class that searches the node's child nodes and, if the requested name is not found, continues to search the descendant nodes until the named node is found, or return `nil` if no such descendant exists.

```
func childNode(named name: String, recursive: Bool = true) -> Node?  
{  
  if let child = childNodes.first(where: { $0.name == name } )  
  { return child  
  } else if recursive {  
    for child in childNodes {  
      if let grandchild = child.childNode(named: name)  
      { return grandchild  
      }  
    }  
  }  
  return nil  
}
```

## Root Motion

To move the character around, we need to update the model matrix of the model's root node over time. This will cause the animation to be applied relative to the world transform of the root node. This motion can be driven by user input, which is commonly used to move characters in 3D games.

In the sample code, we use a little trigonometry to make the character run in a circle. Assuming we have a reference to the character's root node (using the method from the previous section), we compose a translation matrix and rotation matrix together, to perform root motion over time:

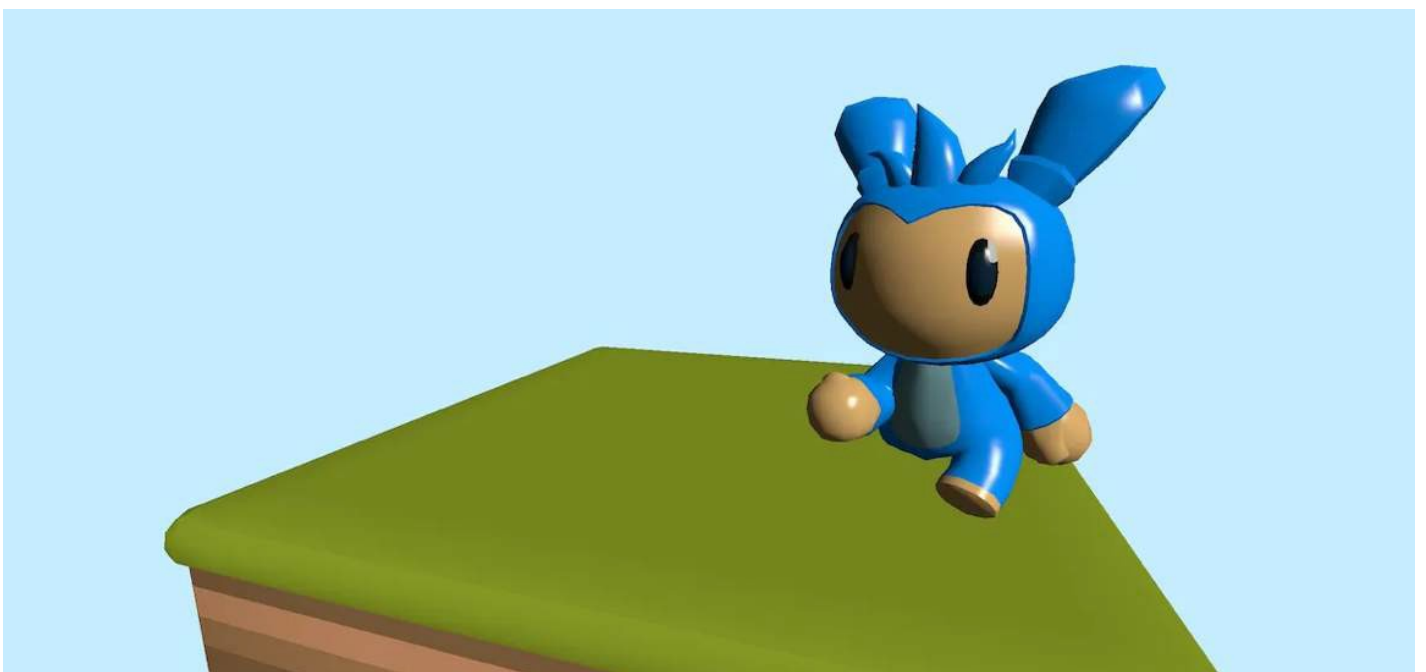
```
// How wide of a circle to run in
let circuitRadius: Float = 3.0
// How long it takes to complete one circular lap
let circuitDuration: TimeInterval = 6.0

let runTime = fmod(time, circuitDuration)
let runAngle = Float((2.0 * .pi * runTime) / circuitDuration)
let position = SIMD3<Float>(circuitRadius * cos(runAngle), 0,
circuitRadius * sin(runAngle))

let rotation = float4x4(rotateAbout: SIMD3<Float>(0, 1, 0),
byAngle: runAngle)
let translation = float4x4(translate: position)

character.transform = translation * rotation
```

With root motion applied, our character is now able to go for a jog around our little world:



In this lengthy article, we have explored how to load and render skinned, animated models with Model I/O and Metal. In the final two articles of this series we will delve into physically-based rendering and postprocessing to add extra degrees of realism to our virtual scenes.

# Day 29: Physically Based Rendering



Warren Moore ·

21 min read · Nov 22, 2022

*This series of posts is my attempt to present the Metal graphics programming framework in small, bite-sized chunks for Swift app developers who haven't done GPU programming before.*

*If you want to work through this series in order, start [here](#). To download the sample code for this article, go [here](#).*

Accurately accounting for the interaction between light and matter is one of the great challenges of computer graphics — especially real-time graphics.

So far, our lighting and material models have been *empirical*: based on observation rather than principle. They have also been highly *non-physical*: they do not conserve energy; materials do not respond realistically to different lighting environments; and material and light properties do not correspond to real-world quantities.

*Physically based rendering* (PBR) is the branch of computer graphics that attempts to build models of light and matter that are grounded in reality. Since the early 1980s, PBR has been an active area of research. New advancements continue to occur each year.

Our approach to physically based rendering will only scratch the surface of this deep and dynamic field. We will first refine our material model so that we can specify materials that correspond more closely to the types of materials we see in the real world. Then, we will introduce some essential topics in physically based rendering. Finally, we will see how to implement physically based rendering with Metal shaders.

This post will use heavier math than we've used before, so feel free to skip

the equations on your first read. You will also find it helpful to consult other resources that describe the theory and mathematics of PBR more thoroughly; I will mention some along the way. Learning physically based rendering takes time, but this article will give you enough knowledge to use existing source code to incorporate PBR into your apps.

## ***Material Models for Physically Based Rendering***

As programmable GPUs entered the consumer market in the late 00's, physically based rendering started to percolate into the field of real-time rendering. Since 2006, the latest developments in the field have been presented at courses such as Advances in Real-Time Rendering in 3D Graphics and Games at the SIGGRAPH conference.

In a 2012 course, Brent Burley of Walt Disney Animation Studios shared a material and lighting model that introduced the now-familiar metallic- roughness parameterization. The following year, Brian Karis presented on how the Disney model had been incorporated into Unreal Engine 4 for real- time use. Our approach here borrows heavily from this work and other contemporary developments.

Some resources and software packages refer to their physically-based shading implementation as *principled*. This term refers in part to the principles used to select a shading model's parameters. I paraphrase the principles enumerated by Burley in 2012 as follows.

1. Parameters should be intuitive, not necessarily “physical.”
2. Use as few parameters as possible.
3. Parameters should range from zero to one. For color parameters, this means each component (RGB) ranges from zero to one.
4. Setting a parameter to a value outside its range should produce plausible results when possible.
5. All combinations of parameter values should produce plausible results.

Note the emphasis on intuition here: this set of principles focuses on the



artist rather than the rendering software. Material parameters often have no physical unit associated with them: a roughness of 1 just means “as rough as possible”, and a metalness value of 1 just means “metal.” Our job as graphics programmers is to translate these artist-friendly parameters into the equations and shader code that produce good-looking pictures.

Our material model will be somewhat simplified from the one implemented in Unreal Engine 4, but after getting acquainted with the theory and practice of physically based rendering, you will be equipped to implement your own shading model that has your desired features.

Our material model for surfaces has three core parameters: metalness, roughness, and base color.

## ***Metalness***

Metalness is the most significant material parameter. Metalness unifies metal and dielectric material calculations in shaders and ordinarily takes a value of 0 (“fully dielectric”) or 1 (“fully metallic”). Intermediate values other than 0 or 1 might occur when blending between materials in a texture map, but in the real world, materials almost always act like metals or dielectrics. This applies even to metal oxides like rust (which is a dielectric) and metallic paint flecks (which can be modeled with specialized shaders as metals embedded in a dielectric medium).

## ***Roughness***

Another core surface parameter is *roughness*. Roughness ranges from 0 to 1, with 0 representing perfect mirror smoothness and 1 representing maximum roughness. This parameter captures surface variations that are smaller than those that are modeled with normal maps. We will see below in the section on microfacet theory how roughness is modeled mathematically.

## ***Base Color***

The final core parameter of our materials is the *base color*. The interpretation of base color depends on the metalness of the material.

For dielectrics, base color is interpreted as *albedo*. Albedo is the proportion of light reflected by a diffuse surface lit uniformly from all directions. It is what we ordinarily think of as the “color” of a non-metallic surface. Metals do not reflect diffusely, so the albedo of a metal is black.

Conversely, for metals, base color is interpreted as the *specular color*. Since metals only reflect specularly, the specular color is what gives metals like aluminum, gold, and platinum their unique tones. As we will see below, dielectrics also reflect specularly, but do not tint the reflected light as metals do.

## ***What’s Missing from Our Material Model***

Our material model ignores a number of physically significant phenomena in favor of simplicity:

- *Anisotropy* is the tendency of surfaces to reflect light along preferred directions rather than uniformly. A common example of a material exhibiting this effect is brushed aluminum. Medium-scale anisotropy can be modeled with a normal map. Anisotropy below this scale can be modeled with specialized reflectance functions.
- *Glossy refraction* occurs when a rough translucent surface *transmits* light rather than reflecting or absorbing it. Ground glass is a substance that produces such refractive effects. Rendering refraction in a raster-based system requires somewhat specialized techniques that are beyond the scope of this text.
- Diffraction and other spectral effects like iridescence are not accounted for, since we treat light as a blend of three primary colors rather than tracking its wavelength. Diffraction can be faked with specialized shaders, and some raytracing systems model it precisely.

Finally, our model is only a solid surface model: we make no attempt to account for subsurface scattering or any optical phenomena that happen *inside* surfaces.

## ***Theory of Physically Based Rendering***

In this section, we will cover numerous theoretical topics in physically based rendering, starting with the microfacet model and proceeding through the construction of a reflectance equation that more accurately reflects reality than our previous efforts.

In the following, we will use some mathematical shorthand below to make formulas more compact.

We define the halfway vector as  $h = \text{normalize}(l + v)$ .

Whenever we want to refer to the angle between the surface normal and another vector  $x$ , we will write  $\vartheta_x$  (“theta-sub- $x$ ”). Sometimes, when we need the cosine of the angle between unit vectors, we write the dot product  $(n \cdot l)$  rather than saying  $\cos(\vartheta_l)$ .

Formulas may depend on material properties like roughness and base color; we omit these as parameters everywhere for brevity.

## ***The Microfacet Model***

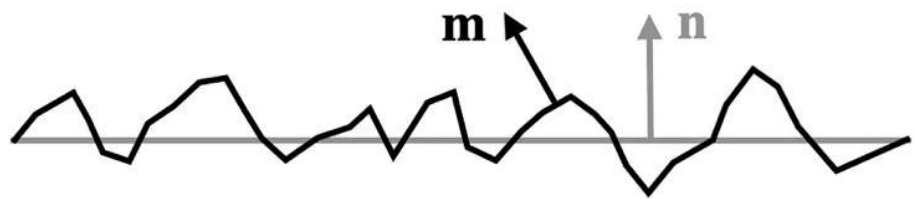
When we care about surface variations smaller than those captured by geometry or normal maps, we envision a surface as a collection of microscopic subfaces called *microfacets*.

If we think of a surface as a continuum divided up into tiny subfaces, roughness tells us the extent to which these subfaces are oriented in differing directions. A perfect mirror — a surface with a roughness near 0 — has microfacets whose normals are all aligned in the same direction. On the other hand, a rough surface is comprised of microfacets whose orientations are more or less random, causing incident light to be reflected in many different directions.

When working with microfacets, we have to consider two different normals in our shading calculations: the *surface normal* (which is the geometric normal possibly perturbed by a normal map) and the *microfacet normal*.

When we need to distinguish between them, we will label the geometric normal  $n$  and the microfacet normal  $m$ .

The figure below illustrates the geometric normal and the microfacet normal of one of the many microfacets that comprise the surface.



The theory of microfacets was introduced to the computer graphics literature in the 1980s by Cook and Torrance. It has become the dominant paradigm for modeling the microscale characteristics of reflective and refractive surfaces in the PBR literature.

The question we always want to answer when rendering 3D scenes is: Given a set of lights and objects, how much (and what color) light reaches our eye from a particular direction? We have already answered this question to a first approximation with our previous lighting models. The Blinn-Phong model evaluated with Phong shading tells us what a surface looks like under idealized conditions.

In order to know how much light is reflected from a given point on the surface, we need to know two things: What amount of light arrives at the surface from a given direction, and how does the surface respond to (i.e. reflect) it?

The intensity of light arriving at a surface (a quantity called *radiance*) can be calculated from the light's properties and the attenuation that occurs between the light and the surface. This intensity is completely independent of the properties of the surface itself, and we will continue using the same light properties and attenuation model in our revamped shading implementation. The improvements we make will be based in more realistic modeling of reflection.

## ***Bidirectional Reflectance Distribution Functions***

Once we know the intensity of light arriving at a surface from a particular direction, we need to determine how the surface responds. This is modeled by a function called the *bidirectional reflectance distribution function*, or BRDF. We write the BRDF with the symbol  $f_r$ .

A BRDF answers the following question: What portion of light arriving at a surface from a given incoming direction is reflected along a given outgoing direction?

As we have done before, we will split up the diffuse and specular parts of surface reflection into different terms. This means that our BRDF will consist of two terms, which we will call  $f_d$  and  $f_s$ :

$$f_r = f_d + f_s$$

We will also refer to  $f_d$  as the “diffuse BRDF” and  $f_s$  as the “specular BRDF.” They are each BRDFs in their own right; they just capture different kinds of reflection.

The Lambertian reflectance model we have used previously is simplistic, but it gives plausible results for many different kinds of dielectric materials, so we will continue using it. Written in BRDF form, it looks like this:

$$f_d = c_d/\pi$$

Here,  $c_d$ , sometimes called the diffuse color, is just the albedo described above as the base color. The factor of  $1/\pi$  comes from the fact that diffuse reflection scatters incoming light uniformly in all directions. It turns out that we need to divide by  $\pi$  to ensure energy conservation in the diffuse term.

The specular BRDF is more complicated than the diffuse BRDF because we need to account for additional information:

- the viewing direction,

- the light direction, and
- the distribution of microfacet normals, which is dependent on the roughness of the surface.

The specular BRDF is proportional to the product of three factors:

$$f_s \propto D \times G \times F$$

These factors have the following meanings:

- $D$  is a microfacet normal distribution function,
- $G$  is a shadowing-masking function, and
- $F$  is a Fresnel reflection function.

Since the specular BRDF is more complex than the diffuse BRDF, we give it its own dedicated section.

## ***The Specular BRDF***

There are many possible choices for the factors in the specular BRDF, each with their own tradeoffs in accuracy, computational cost, and complexity. I selected these factors based on ease of implementation. As you go about implementing your own renderer, you may encounter situations where other choices make better sense for the kinds of pictures you want to produce.

## ***Remapping Roughness***

For the sake of numerical accuracy, we will remap the roughness values in our materials to the range 0.08–1.0 in our shaders. Perfectly smooth surfaces do not exist in reality, and some of the functions we will use in our shading equations have singularities when roughness is 0: their value becomes infinite. For this reason, we boost the minimum roughness just a little bit. In pseudocode, this looks like:

```
roughness = remap(0.0, 1.0, 0.08, 1.0, materialRoughness)
```

Furthermore, it turns out that roughness is not perceptually linear. If we take our remapped roughness value and use it directly, surfaces will seem to become too rough too quickly as roughness increases. So, we apply another mapping, squaring the roughness to get a constant we will use in many of our subsequent calculations. We call this constant  $\alpha$  (the lowercase Greek letter alpha):

```
alpha = roughness * roughness
```

This alpha is unrelated to the alpha we use to represent surface coverage when discussing transparency and alpha blending.

## ***The Normal Distribution Function***

At the microscopic level, we consider every microfacet to be a perfect, mirror-like reflector. This means that light is either reflected totally by a microfacet, or not reflected at all. In other words, the reflectance of an individual facet is *binary*. Reflection happens when the microfacet normal aligns exactly with the half-way vector ( $h$ ) between the light direction and the view vector.

At the macroscopic level, we don't want to consider microfacets individually, as this would be very computationally expensive. Instead, we consider their aggregate behavior using statistical methods. This is similar to how we use the single quantity "temperature" to summarize the random, microscopic movements of molecules.

Since we are treating microfacets as a statistical population rather than as individuals, we will use a function that calculates the *statistical distribution* of microfacet normals that align with the macroscopic surface normal. This function is symbolically written  $D$ .

This distribution is highly dependent on surface roughness. When roughness is low, it is clustered tightly near the origin, giving rise to sharp, mirror-like reflections. Conversely, when roughness is high,  $D$  approaches a constant value, which geometrically means that the microsurface normals are randomly distributed.

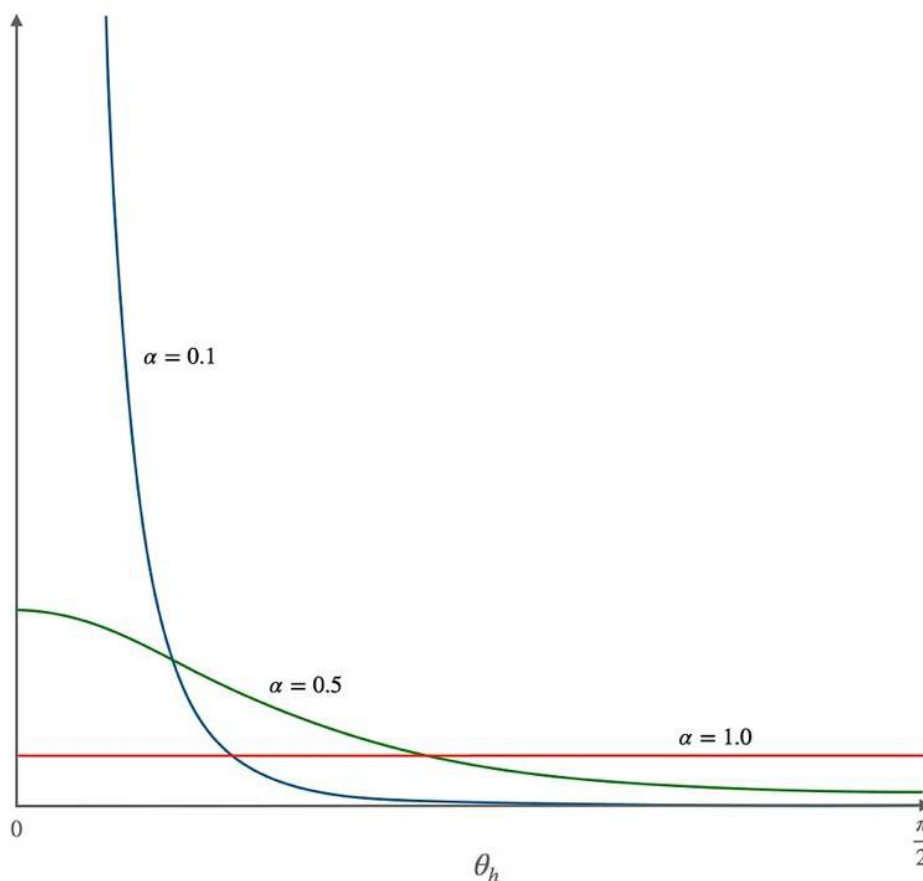
We will not go through a full derivation of normal distribution functions, but the 2007 EGSR paper by Walter, et al, provides a good survey, along with a handful of alternative distribution functions. One popular choice for  $D$  was published by Trowbridge and Reitz in 1975:

$$D(\mathbf{n}, \mathbf{h}) = \frac{\alpha^2}{\pi} \frac{1}{[(\alpha^2 - 1)\cos^2\theta_h + 1]^2}$$

For moderate values of roughness, this function gently slopes from its maximum value at  $\vartheta_h = 0$  toward 0 as  $|\vartheta_h|$  approaches  $\pi/2$ . For more extreme values of roughness (near 0 or near 1), it behaves as described above: steep and narrow at low roughness, flat at high roughness. The exact values of the function are less important for our purposes than its overall shape.

The figure below illustrates the shape of  $D$  for a few different roughness values (0.1, 0.5, and 1.0).



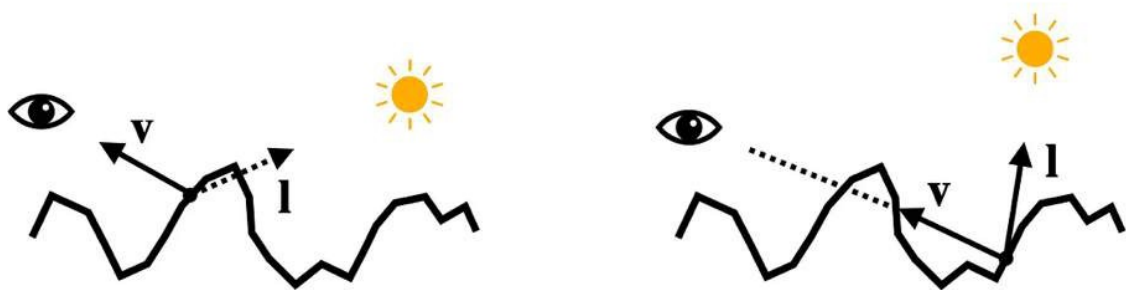


## The Shadowing-Masking Function

The next factor that contributes to the specular BRDF is the *shadowing-masking function*, which we represent with the symbol  $G$ .

When we say “shadowing,” we mean light being physically blocked from *arriving* at a particular microfacet by other nearby microfacets. When we say “masking,” we mean light being blocked *after reflecting* from a particular microfacet. Since we don’t account for light that eventually reaches the viewer after reflecting more than once, we consider masked light to be lost, which can produce artificially dark pictures. Either way, both of these phenomena — masking and shadowing — are important in accounting for how much light actually makes it to the viewer from a light source.

The figure below illustrates shadowing and masking. In the shadowing case (left), the light ray is not reflected from the point of interest because it is blocked by other microsurface features. In the masking case (right), the light does not reach the viewer because another part of the surface blocks the already reflected ray.



It turns out that masking and shadowing are fundamentally the same phenomenon. The only difference is which perspective we are looking from: the light's or the viewer's. Suppose we could write a formula to describe the proportion of light *not* masked by the microstructure of a surface (in other words, the fraction of light that reaches the viewer assuming it has *already reflected* from the surface). This formula could then be used to calculate shadowing and masking, by changing the parameters we use to evaluate it.

As part of our shadowing-masking calculation, we will use a formula presented by Smith in 1967. In the computer graphics literature, it is commonly written with the symbol  $G_1$ :

$$G_1(\theta) = \frac{2}{1 + \sqrt{1 + \alpha^2 \tan^2 \theta}}$$

Here,  $\vartheta$  is the angle between the surface normal  $n$  and the other vector of concern ( $l$  for shadowing and  $v$  for masking). Assuming masking and shadowing are independent, the total proportion of light reaching the viewer from the light is the product of two evaluations of  $G_1$ . This gives us the final form of the  $G$  factor for our specular BRDF:

$$G(\mathbf{n}, \mathbf{l}, \mathbf{v}) = G_1(\theta_l)G_1(\theta_v)$$

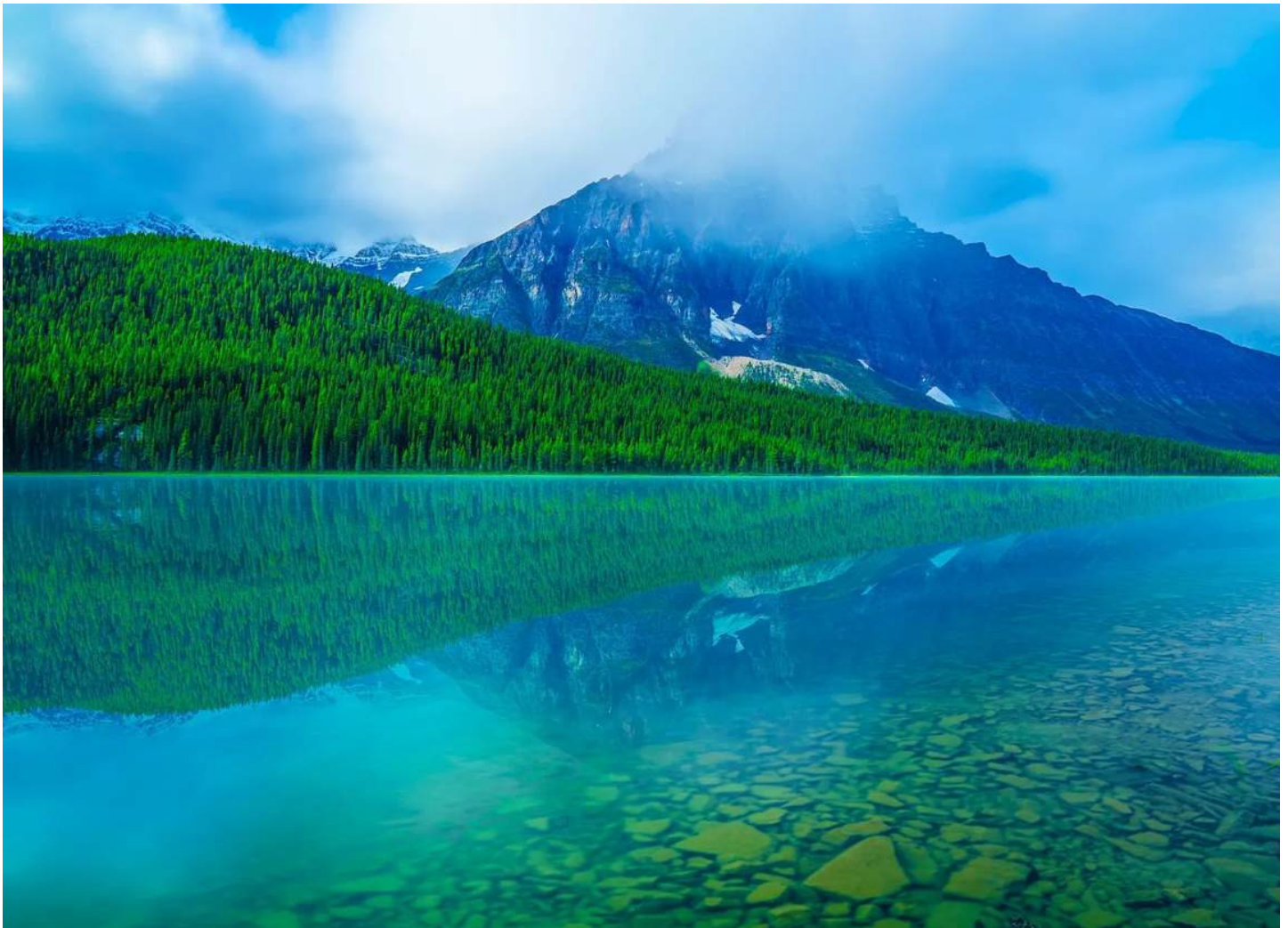
For a thorough explanation and derivation of shadowing-masking functions, consult Heitz's 2014 JCGT paper.



## ***Fresnel Reflection***

Regardless of material, as the view direction approaches perpendicularity with the surface normal, reflection becomes mirror-like. This phenomenon is called *Fresnel reflection*.

The figure below illustrates Fresnel reflection on the surface of a still lake. In the foreground, the viewing angle is steep relative to the surface normal, and the water is nearly transparent (i.e. minimally reflective). In the distance, as the viewing angle becomes shallower, the water reflects the distant scenery like a mirror.



*Photo by [Joshua Woroniecki](#) on [Unsplash](#)*

Specular reflection behaves differently for dielectrics and metals. Reflections from dielectric surfaces are not “tinted” by the color of the surface; we say that such reflection is *achromatic*. On the other hand, reflections from metal surfaces are tinted by the color of the metal (they are *chromatic*).

Our Fresnel factor will be based on the specular reflectance as measured

when the incident vector is equal to the normal. This specular reflectance is what we mean when we say “specular color.” We represent this color with the symbol  $F_0$ . For dielectrics, this value is very small. When evaluating Fresnel reflectance of a dielectric, we will use a constant value of 0.04.

A full expression of Fresnel reflection depends on the index of refraction of the surface and the surrounding medium. We will not include these in our Fresnel factor explicitly. Instead, we use a formula published by Schlick in 1994, which depends only on the specular color, the viewing direction, and the halfway vector:

$$F(\mathbf{v}, \mathbf{h}) = \mathbf{F}_0 + (1 - \mathbf{F}_0)(1 - |\mathbf{v} \cdot \mathbf{h}|)^5$$

Schlick’s formula does a good job of cheaply modeling Fresnel reflection for a wide variety of materials.

## ***Assembling the Specular BRDF***

We have covered the essential parts of our specular BRDF:  $D$ ,  $G$ , and  $F$ . We can now assemble them into the complete specular BRDF:

$$f_s(\mathbf{n}, \mathbf{l}, \mathbf{v}) = \frac{D(\mathbf{n}, \mathbf{h})G(\mathbf{n}, \mathbf{l}, \mathbf{v})F(\mathbf{v}, \mathbf{h})}{4(\mathbf{n} \cdot \mathbf{l})(\mathbf{n} \cdot \mathbf{v})}$$

The factors in the denominator arise from some particulars of the microfacet model and are explained by [Nayer, et al 1991](#) (Appendix D) and illustrated with an alternative explanation by [Stam 2001](#) (Appendix B). We will not discuss them further here.

## ***The Rendering Equation***

Now that we have found the diffuse and specular components of our BRDF, we are ready to use them together to calculate the light reflected by a single surface point. For this, we will use a simplified version of an equation called the *rendering equation*.

The rendering equation was introduced by Kajiya at SIGGRAPH 1986. It answers the question: For a given point that lies along the viewing direction, what quantity of light arrives at the viewer, taking into account all of the light arriving at the point or emitted by the surface at the point? The phrase “all of the light” tells us that we need to account for all possible incoming directions. Practically speaking, we need to perform *numerical integration* to gather the incoming light and evaluate the BRDF for each such direction, adding up the contributions to find the total quantity of reflected light.

However, this isn’t as hard as it sounds, because each analytic light (whether a directional light or omnidirectional light) only arrives along a single direction. Therefore, what would be an integral turns into a sum:

$$L_o = L_e + \sum_k L_k \left[ \frac{c_d}{\pi} + \frac{D(\mathbf{n}, \mathbf{h})G(\mathbf{n}, \mathbf{l}, \mathbf{v})F(\mathbf{v}, \mathbf{h})}{4(\mathbf{n} \cdot \mathbf{l})(\mathbf{n} \cdot \mathbf{v})} \right] (\mathbf{n} \cdot \mathbf{l})$$

Here,  $L_o$  is the total reflected radiance,  $L_e$  refers to light emitted from the surface, and  $L_k$  refers to the incoming radiance of the current light. The  $\Sigma$  indicates that we loop over our lights and add up the results. For each light, we use our diffuse and specular BRDFs to compute the intensity of reflected light.

## ***Implementing Physically Based Rendering in Metal***

Implementing our physically-based shading model in Metal consists of two main activities. First, we need to load some new material properties from our 3D assets using Model I/O. Second, we need to modify our shaders to use these new properties to implement our more plausible lighting.

## ***Physically Based Materials in Model I/O***

Modern 3D asset formats like glTF 2.0 and USD support physically-based materials. We will be using USDZ models in our sample project, since it is the format best supported by Apple.

If you happen to have legacy assets in other formats, you can use the Reality



Converter app provided by Apple to convert them to USDZ. Reality Converter’s simple drag-and-drop user interface allows you to specify texture maps for the material properties supported by the format. The figure below shows one of the assets from the sample project in Reality Converter. The property inspector on the right side of the window shows the individual material properties as texture maps.



*Screenshot of Reality Composer showing a sample asset with various physically based material properties*

In code, we will continue to use Model I/O to import assets, including materials. We will need to upgrade our material system to support the new physically based parameters we want to use in our physically based shaders.

We begin by expanding our `Material` class with our principled material parameters. Each parameter can be driven by a constant or a texture map, which allows each property to vary across the object’s surface.

```
class Material {
    var baseColorTexture: MTLTexture?
    var baseColor = SIMD4<Float>(1, 1, 1, 1)
    var emissiveTexture: MTLTexture?
    var emissiveColor = SIMD4<Float>(0, 0, 0, 1)
    var metalnessTexture: MTLTexture?
    var metalnessFactor: Float = 0.0
    var roughnessTexture: MTLTexture?
    var roughnessFactor: Float = 0.5
    var normalTexture: MTLTexture?
    var occlusionTexture: MTLTexture?
```





```

    var opacity: Float = 1.0
    ...
}

```

In addition to the now-familiar base color, metalness, and roughness, we store an ambient occlusion factor, which approximates the degree to which the surface itself occludes indirect light.

We load these material properties with `MTKTextureLoader` as we have before.

To pass material properties into our shaders, we need to use a combination of constants and parameters. Our updated material constants structure looks like this:

```

struct MaterialConstants
{
    float4 baseColorFactor;
    float4 emissiveColor;
    float metalnessFactor;
    float roughnessFactor;
    float occlusionWeight;
    float opacity;
};

```

The updated signature of our fragment shader illustrates how we pass the material parameter textures:

```

fragment float4 fragment_main(FragmentIn in [[stage_in]],
    constant FrameConstants &frame [[buffer(FragmentBuffer::fr
    constant Light *lights [[buffer(FragmentBuffer::li
    constant MaterialConstants &materialProperties [[buffer(FragmentBuffer::ma
    texture2d<float, access::sample> baseColorTexture [[texture(FragmentTexture::
    texture2d<float, access::sample> emissiveTexture [[texture(FragmentTexture::
    texture2d<float, access::sample> normalTexture [[texture(FragmentTexture::
    texture2d<float, access::sample> metalnessTexture [[texture(FragmentTexture::
    texture2d<float, access::sample> roughnessTexture [[texture(FragmentTexture::
    texture2d<float, access::sample> occlusionTexture [[texture(FragmentTexture::

```

## Physically Based Shader Preliminaries

The BRDF is only part of the implementation of a physically based shader. We also need to retrieve each of the material properties from its associated texture map, or fall back onto a constant factor in the case where no texture is available for a property. We will use a simple structure in MSL to hold our material properties:

```
struct Material
{
    float4
    baseColor; float
    metalness; float
    roughness;
    float ambientOcclusion;
};
```

We begin by sampling the various material textures and establishing a value for each material property:

```
float ambientOcclusion = is_null_texture(occlusionTexture) ? 1.0f :
    mix(1.0f, occlusionTexture.sample(repeatSampler, in.texCoords),
    float4 baseColor = is_null_texture(baseColorTexture) ? materialProperties.baseC
    baseColorTexture.sample(repeatSampler, in.texCoords) * mater
float authoredRoughness = is_null_texture(roughnessTexture) ? materialPropertie
    roughnessTexture.sample(repeatSampler,
    in.texCoords). float metalness = is_null_texture(metalnessTexture) ?
    materialProperties.metaln metalnessTexture.sample(repeatSampler,
    in.texCoords).b * mate
```

We create an instance of the `Material` structure and assign these values to it, including the remapped roughness discussed above:

```
Material material;
material.baseColor = baseColor;
material.roughness = remap(0.0f, 1.0f, 0.045f, 1.0f, authoredRoughness);
material.metalness = metalness;
material.ambientOcclusion = ambientOcclusion;
```

## Implementing the Diffuse BRDF

Implementing the diffuse BRDF is relatively straightforward. We know that



diffuse reflection scatters incoming light uniformly out of surfaces in every direction, so we don't need to include any directionally dependent dot products in our diffuse BRDF.

Remembering that we need to include a factor of  $1/\pi$  to account for the fact that we will be integrating over all directions surrounding the surface point, we translate the formula for the diffuse BRDF into Metal Shading Language as follows:

```
float3 Lambertian(float3 diffuseColor)
{ return diffuseColor * (1.0f / M_PI_F);
}
```

where the `diffuseColor` parameter is the albedo of the surface.

## ***Implementing the Specular BRDF***

Implementing the specular portion of the BRDF is much more involved than the diffuse portion. Even so, we will be able to implement it by translating each specular BRDF factor into its own function, then assembling them together into the full specular BRDF.

To begin with, we will translate the normal distribution function. Since we selected Trowbridge and Reitz's formulation, our `D` factor will be computed by calling a function called `D_TrowbridgeReitz`. We implement it as follows:

```
float D_TrowbridgeReitz(float alphaSq, float NdotH)
{ float c = (NdotH * NdotH) * (alphaSq - 1.0f) + 1.0f;
  return step(0.0f, NdotH) * alphaSq / (M_PI_F * (c * c));
}
```

It may not be immediately obvious how this function corresponds to the formula for Trowbridge and Reitz's  $D$  function, but with some inspection, hopefully you can see how it computes the expected result. Note that we use

a step function to force the value of  $D$  to zero when  $n \cdot h$  is less than zero; this corresponds to the situation where the halfway vector points beneath the macro-surface.

Continuing, we will look at how to evaluate the shadowing-masking function,  $G$ . Just as we separated  $G$  into two factors, we will first write a shadowing-*or*-masking function, then use the product of two evaluations of it to find the complete  $G$  factor.

Our  $G_1$  function looks like this:

```
float G_1(float alphaSq, float NdotX)
{ float cosSq = NdotX * NdotX;
  float tanSq = (1.0f - cosSq) / max(cosSq, 1e-4);
  return 2.0f / (1.0f + sqrt(1.0f + alphaSq * tanSq));
}
```

This formulation of  $G_1$  looks somewhat different from the formula above. To avoid having to compute the half-vector angle explicitly, we use a trigonometric identity to find  $\tan \vartheta$  more directly. The rest of this function is a direct translation of the formula for  $G_1$ .

With  $G_1$  in hand, we can construct our full geometric shadowing-masking function:

```
float G_JointSmith(float alphaSq, float NdotL, float NdotV)
{ return G_1(alphaSq, NdotL) * G_1(alphaSq, NdotV);
}
```

The final piece of the specular BRDF is the Fresnel factor. Using the value of  $F_0$  as previously defined, the Fresnel reflectance is a function of  $F_0$  and  $v \cdot h$ :

```
float3 F_Schlick(float3 F0, float VdotH) {
    return F0 + (1.0f - F0) * powr(1.0f - abs(VdotH), 5.0f);
}
```

With the various pieces implemented, we can now construct the full specular BRDF. Assuming that we are in the light loop and have the various dot product values available, along with `F0` and `alpha`:

```
float D = D_TrowbridgeReitz(alphaSq, NdotH);
float G = G_JointSmith(alphaSq, NdotL, NdotV);
float3 F = F_Schlick(F0, VdotH);

float3 fs = (D * G * F) / (4.0f * abs(NdotL) * abs(NdotV));
```

The code above is a direct translation of the specular BRDF formula, using the functions we just defined.

## Evaluating the Total BRDF

To evaluate the full BRDF, we will write a function that takes the material and the various dot products computed in our light loop, and returns the total surface reflectance:

```
float3 BRDF(thread Material &material,
            float NdotL,
            float NdotV,
            float NdotH,
            float VdotH)
{
    float3 baseColor = material.baseColor.rgb;
    float3 diffuseColor = mix(baseColor, float3(0.0f), material.metalness);
    float3 fd = Lambertian(diffuseColor) * material.ambientOcclusion;
    float3 F0 = mix(DielectricF0, baseColor, material.metalness);
    float alpha = material.roughness * material.roughness;
    float alphaSq = alpha * alpha;
    float D = D_TrowbridgeReitz(alphaSq, NdotH);
    float G = G_JointSmith(alphaSq, NdotL, NdotV);
    float3 F = F_Schlick(F0, VdotH);
    float3 fs = (D * G * F) / (4.0f * abs(NdotL) * abs(NdotV));
    return fd + fs;
}
```



To incorporate this function into our fragment shader, we will continue to use a loop to calculate the contribution of each light. First, we determine the light direction and light intensity as usual. Then, we calculate the various light-dependent and view-dependent dot products. Finally, we accumulate the contribution of the light by multiplying the light intensity by the BRDF, giving the intensity of the light from the current light reflected by the current surface point.

```
for (uint i = 0; i < frame.lightCount; ++i)
{
    Light light = lights[i];
    float3 lightToPoint = light.directionToPoint(in.eyePosition);
    float3 intensity = light.evaluateIntensity(lightToPoint);
    float3 L = normalize(-lightToPoint);
    float3 H = normalize(L + V);
    float NdotL = dot(N, L);
    float NdotV = dot(N, V);
    float NdotH = dot(N, H);
    float VdotH = dot(V, H);
    surface.reflected += intensity * saturate(NdotL) *
                          BRDF(material, NdotL, NdotV, NdotH, VdotH);
}
```

The outgoing radiance is the sum of the surface's emitted and reflected radiance, so we find the final surface color like so:

```
float3 color = surface.emitted + surface.reflected;
```

## Results

I have used Reality Composer to build a set of USDZ assets that showcase the various parameters of the physically based shading model we have implemented. We with same shader, we can get decent rendering of such diverse materials as metal, plastic, glazed ceramic, and wood, as shown below.





These enhancements go a long way toward increasing the realism of our scenes, but still leave much to be desired. Since our lights are still point- like, the fidelity of reflections doesn't match the richness of the real world, so our metals wind up looking somewhat drab.

In the next entry, we will look at image-based lighting, a technique that uses cubemaps to represent all of the light coming from the environment surrounding the scene. Image-based lighting is the last crucial stop of our journey to rendering realistic pictures.